Predicting vulnerable files by using machine learning method

Xiwei Shen

Master of Science Thesis

Delft University of Technology

Delft Center for Cyber Security

Predicting vulnerable files by using machine learning method

Master of Science Thesis

For the degree of Master of Science in Computer Science at Delft University of Technology

Xiwei Shen

September 20, 2018

Faculty of Electrical Engineering, Mathematics and Computer Science(EWI) · Delft University of Technology

The work in this thesis was supported by eBay classified group Incorporated. Their cooperation is hereby gratefully acknowledged.

Table of Contents

Abstract

Web applications have been gaining increased popularity around the globe, in such a way that a growing number of users are attracted to make use of the functionality and information provided by these applications. While providing solutions to complicated problems in a fast and reliable way is one of the most advantages of using web applications, these platforms can cause adverse effect on user's life if controlled in unauthorized way by malicious people. A platform with more vulnerabilities are more likely to be attacked. This research is focusing on building a prediction model for detecting vulnerabilities of web applications at eBay. Based on the analysis of important features, we dig deeper to find decisive factors of web application vulnerabilities. Making use of data on GitHub, we extract features related to source code files and developer networks, such as modification frequency, number of involved developers and duration between two commits. By applying machine learning techniques in the field of vulnerability prediction, we are able to provide reasonable suggestions for developers in the beginning phase. This can help develop relative defect-free and well-documented software. In this paper, we will explain the prediction model in detail from the aspects of code complexity, developers' behaviors and their networks. Moreover, according to results of various classifiers, we offer possible causes of vulnerabilities and reasonable suggestions for avoiding vulnerabilities in the future. To conclude, main contributions of this thesis are valuable feature engineering, the machine learning model and applicable suggestions for predicting vulnerabilities effectively at eBay.

Keywords Machine learning, imbalanced learning, network theory

Preface

This thesis is the final work of my master study in Computer Science at TU Delft. The thesis project is performed at eBay as a nine-month internship. The process of production of my thesis work is quite interesting and enjoyable. I would like to express my gratitude to the following people for their help during my thesis project. My dear supervisors Sicco Verwer, who helps me to find this internship opportunity and help me find new research directions when I am lost. Thanks to Pieter Hartel and Saeed Sedghi, without their professional guidance and suggestions, I could not produce so much wonderful work in this research. Dr. Maurício Aniche, who is so nice helping with my graduation process and helping me a lot with the tool that I used to collect GitHub log files, and he is so kind to be my committee member. And my colleague Arnoud Witt, Jack Schilder and Nico Mossel, they are the kindest people I have met here. Without their assistance and support, this internship would be much difficult.

I would also like to thank all my dear friends. Their support and their accompany makes my life much more happy. Besides, I would like to give my thanks to my best friend, Hanxi Wang. With her encouragements from China, I feel fulfilling all the time.

At last, I would like to express many gratefulness to my dear parents, who support my decisions all the time and always give their best wishes to my life!

Xiwei Shen
September, 2018

Chapter 1

INTRODUCTION

1-1 BACKGROUND INFORMATION

Nowadays, web applications are widely used in many of our daily activities on the Internet. More and more Hackers now work on hacking these web applications by putting efforts into finding available vulnerabilities exposed in web applications. A web application vulnerability is a hole or weakness in the application that allows an attacker to do harm to application owner, users and other entities that rely on the application. Normally, a vulnerability is caused by a design flaw or an implementation bug. How to detect and fix web application vulnerabilities is a fundamental and important problem for e-commercial platform runners to concern about. In order to help reduce the number of vulnerabilities caused by implementation bug during the development process, most Internet companies apply security check tools or get help from third-party security testing companies. Both these security check tools and security check companies do almost the same thing: using an existing regulation library to do testing on applications and then generate reports. There are also some differences between them: security check tools have the full access to source code, so they are able to detect existing vulnerabilities and their corresponding code file or code line; Security testing company can only point out which web page has which kind of vulnerability and generate reliable vulnerability reports since they do testing work on client side.

The whole security process is shown below: First, developers develop platforms. Second, doing regularly security check on their implemented platform and receiving security reports from security analysis tools or security check companies. Third, storing security reports into an internal system and assigning tasks to different groups. Forth, responsible developers do modifications on the source code to fix a specific implementation bug based on security reports. At last, security check companies would do validation on each bug and generate a new report to show the state of these bugs. This procedure is proven to be favorable for reducing the number of vulnerabilities existed in each platform. However, there are also drawbacks. Normally, the whole process of detecting vulnerabilities and fixing them might take over months. Developers are required to do a large amount of repeating work and this is expensive. Meanwhile, none of these steps in the procedure can prevent developers from creating web application vulnerabilities.

This project is aiming at achieving two goals in the end. First, finding main causes of web application vulnerabilities from perspectives of code characteristics and developer factors. Code characteristics can be achieved from source code files, and developer factors are quantified by analyzing Github commits history. Second, intuitively describe what do those feature values indicate. Based on statistical data analysis, giving reliable suggestions for development group in order to prevent making vulnerabilities from the very beginning. Third, building a vulnerability detection model by using a machine learning technique.

1-2 PROBLEMS

The research on vulnerability detection is never stopped. Various tools are created to help developers find vulnerabilities during their application development, and most of them are static analysis tools. The major shortage of static analysis tools is that they are more likely to provide false positive vulnerable results. Despite, the static analysis tool is only able to do alarm on having vulnerable source code line or file. Although these tools have a good performance on having a good overview and understanding of source code files, including security check functions and package designed structure, they do not have access to learn how developers behaved during the application development, but human factors are directly related with the appearance of vulnerabilities. Due to the widespread of version control system Git, it is possible to trace developer work from beginning till the end.

1-3 RESEARCH QUESTION

For the sake of achieving these goals mentioned in section 1, this thesis would focus on introducing methods to extract features about source code complexity, developer behavior, and developer network. To structure this research project and select the research direction to proceed in each stage, several research questions are set up and will be answered based on the experiment results.
For completing this research step by step, three research questions are defined in the first place:

- How to quantify developer works on each source code file from developer working log files?
  By learning from developer working log files, we can focus on studying the aspects of extracting features from the developer network and calculating developer working pattern by going through all the commit history. The method to build a complete developer network will be described in Chapter 3.

- What suggestions can be made to adjust developer behaviors based on collected metrics by using data analysis method?

In this big data era, data-driven strategies are much more reliable and efficient since a result is supported by a large amount of history data. This research also emphasizes the importance of having results from analyzing data, and this research will mainly use data visualization method to support my conclusion because it directly shows the difference between features and classes.

- Is code complexity, developer behavior and developer network metrics important for detecting web application vulnerabilities? Which one is most important to distinguish vulnerable files from normal files?
  Based on the study on accessible data resource, I select three aspects to study further. The feature collection procedure is a basic and significant procedure for building a whole machine learning project. To make this project meaningful, project result should be instructive. For instance, if code complexity is the killer of application safety according to the result, developer groups should be reminded of avoiding making complex code file.

In conclusion, these research questions are settled to guide the procedure of building a prediction model for web application vulnerabilities. At the same time, analyzing collected features to find main causes of vulnerabilities, then giving suggestions for developers to adjust their behavior. At last, evaluating and proving the value of this research.

1-4 RESEARCH SCOPE

This research involves several areas, One part of these areas is data engineering on accessible raw data including data collection, data pre-processing, feature measurement and data labeling. Another part is about classification by different machine learning technique, and imbalanced learning is specially introduced to this project due to the characteristic of the collected data set. Despite, this research also contains a comparison between different evaluation criterion and data analysis methods to answer my research questions. Among these different parts, feature measurement quality is the dominant factor that would influence the performance of my designed prediction model.
In this research, I study all aspects of a program developer and quantify their contribution to each file and their role in each development group based on the study result. After having the input data for machine learning scheme, I only choose to use the traditional machine learning classifier instead of using neural network or deep learning technique because my available data set size is rather small. For making my prediction model performs better on predicting vulnerabilities, more efforts are paid to tune classifier parameters and select the best-performed classifier by testing and comparing different classifiers. For supporting the experimental result, more result analysis would be made at last.

1-5 RESEARCH CONTRIBUTION

In this thesis, we provide a solution based on the prediction model that aims at discovering the behavior of the developers into security posture of products. Our solution provides a model that predicts security vulnerabilities based on the application development behavioral features, such as the duration and rate of modifying application source codes, using machine learning techniques. The contributions of our vulnerabilities prediction model are as follows:

- Provide insightful information on the impact of the behavior model of the application model into the security vulnerabilities.
- Provide root causes of appearing vulnerabilities in the applications.
- Provide insight on how to adjust features and development parameters in such a way that vulnerabilities are can be considerably reduced.
- Provide visibility on how to allocate and prioritize technology resources on development platforms based on the vulnerability prediction model

1-6 READING GUIDE

In Chapter 2, first introducing some basic knowledge related with this research, then having reviews on related papers of prediction models for vulnerability and implementation bug and also point out which part is used for this research. Chapter 3 is all about the input data of this research, including introducing data resources, details of how to calculating defined features. The experimental procedure is shown in Chapter 4, starting with proposing my designed framework, to describe all technical details during each step. This chapter aiming at showing the performance of this model is reliable. In Chapter 5, After recapping the whole research, I will make some discussions on the result and classifier performance, also present feature relation figures to make suggestions for developers on their daily work. Furthermore, listing future works in this research area.

Chapter 2

LITERATURE REVIEW

This chapter will have a review of papers about detecting software failure or vulnerabilities and how machine learning techniques can be used in the security area. I will first introduce some papers Quite a number of researchers have already made efforts on studying how to detect injection attack risk hole in web applications from different aspects.

This Chapter consists following parts: The first section gives an overview of machine learning techniques, especially supervised learning algorithms, which is related to this research. The second section introduces how to using machine learning techniques to do software faults prediction, also propose the importance of having an imbalanced learning method into consideration when building a fault prediction model in Section 3. In the last section, I highlight several papers building vulnerability prediction models from different aspects, and according to their study object, I categorize these researches into two types, including code file level analysis and program statement level analysis. Despite looking into their design of procedure, I mainly focus on discussing their selected features that are used in the model.

2-1 MACHINE LEARNING

Machine learning technique is widely used for data analysis to build prediction models. According to recent papers, machine learning techniques, which are widely used these days, can be divided into three categories: supervised learning, unsupervised learning and reinforcement learning. To conclude, a supervised learning method can only learn from labeled training data, and on the contrary, unsupervised learning does not require the access to the label of data. Especially, reinforcement learning does not have a restriction on using labeled and unlabeled data. This method is designed to learn from feedback that is retrieved from its interaction with the environment. After considering the advantages and disadvantages of different types of machine learning methods, I decided to use supervised learning for this research. Supervised learning algorithms can be used to train a model of class labels distribution, and this model is able to predict class labels for testing instances. An example of supervised learning algorithms process flowchart is shown in Figure 2-1, this whole process is also called classification. This is the foundation of my designed prediction model as well. It is essential to select which

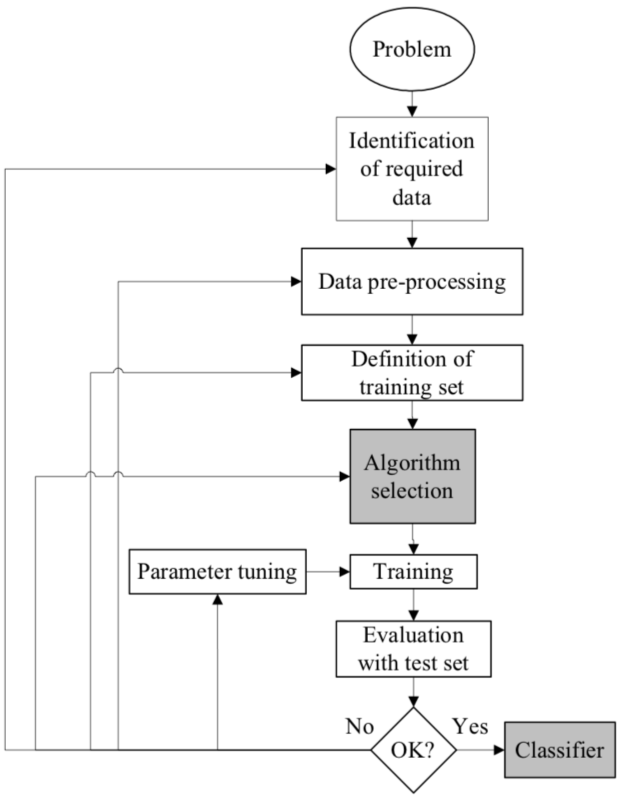


Figure 2-1: The process of supervised learning[1]

classification method to use for a certain problem. There is a review on several widely used supervised learning algorithms in [1]. To decide which classifiers are more suitable for this research, first I look into their pros and cons. In paper[1], the author pointed out that comprehensibility of Decision Tree make this classifier helpful for understanding why an instance is assigned to a certain class, and Decision Tree is a suitable choice when dealing with discrete features. Linear Discriminant Analysis(LDA) and Naive Bayes are both statistical learning algorithms, which can provide a probability about labeling an instance. Moreover, in order to meet the requirement of this research, accuracy, tolerance to noise, the risk of being overfitting and explanation ability are some vital aspects to consider when selecting classifiers. These models are considered in this research.

2-2 MACHINE LEARNING FOR SOFTWARE FAULT PREDICTION

Most classifiers expect equal distributed class data, unfortunately, vulnerabilities detecting training data are more likely to be imbalanced data, since there are always more neutral files existed than vulnerable files in a development project. This directly resulted in having imbalanced data for learning.

Imbalanced learning result is not reliable, because imbalanced learning can make a classifier compromise on achieving better performance instead of predicting its actual label. He et al.[2] summarized different kinds solutions for imbalanced data. The sampling method is a way to do acceptable modifications on imbalanced data in order to achieve a balanced distributed data, such as oversampling and undersampling. Another way to deal with imbalanced data pursue creating a balanced distributed data, instead, setting cost matrices to describe the costs for misclassifying a data, this kind of method is called a cost-sensitive method.

Wang et al. in their paper[3] emphasized that a well-designed software defect prediction should have a balance between defect detection rate and overall performance. It is the same for vulnerability detection.

2-3 IMBALANCED DATA LEARNING

Because of the class distribution skew problem, it is a crucial issue to deal with imbalanced data learning in this research. According to the study on existed techniques, there are several methods to deal with the imbalances:

Assessment Metrics

For assessing the performance of the classifier on training and testing balanced data, previous researches used accuracy and error rate as an important evaluation metrics. However if the classifier can assign all instance into the majority class, the accuracy value could be still high, but it is not an ideal classifier we want to have. Thus, this research chooses to use precision, recall, G-mean, balance, ROC curve and AUC value. Intuitively, precision represents exactness about labeling correctly. On the contrary, recall is a measure of how many instances belong to the positive class were labeled correctly. G-mean value is expressed as the geometric mean of recall values of both the positive and negative classes, a classifier with a good performance should have higher G-mean value. Balance is retrieved by measuring the distance between a certain (precision, recall) point to the ideal point on the ROC curve which is (0,1). ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) with various discrimination thresholds. AUC measures the area under the curve, which can be used as a good evaluation criterion.

Sampling Techniques

Overall, there are two methods suitable for sampling imbalanced data, including random under-sampling and random oversampling. The idea of these two sampling methods is randomly adding(removing) a randomly selected dataset from minority(majority) class to make the whole set becoming balanced. However, these random sampling methods have some shortages. The under-sampling method would cause information loss to majority class, and the oversampling could bring about the over-fitting issue on minority class. Due to a limited number of vulnerable files, this research only uses random oversampling on the dataset.

Generating Synthetic Data Samples

Methods of generating synthetic data samples are aiming at creating new data samples to solve the over-fitting issue. Among existed methods, SMOTE[4] and ADASYN[5] are widely used within imbalanced learning area. By using SMOTE method, the minority class is over-sampled by adding new synthetic samples along the line segments joining any/all of the k nearest neighbors of each sample in minority.[4] But SMOTE method cannot avoid creating overlapping samples, ADASYN is another method to prevent having an overlapping problem. This method would create new data samples based on minority class distribution. Despite, there is a research[6] on combining SMOTE and standard boosting method to utilize their ability to deal with unbalanced data.

2-4 VULNERABILITY PREDICTION AND DETECTION

According to the study object of their researches, we can categorize them into two kinds:

2-4-1 Code File Level Static Analysis

This paper[7] drew a conclusion that data mining static code file attributes are useful for detecting web application vulnerabilities. They argued that whether to use complexity metrics during building prediction models are meaningless, since how to use these metrics are more essential than select which metrics to use. It raised two important issues while building the model.

- In a prediction model, when target detecting class (here it means the vulnerable class) is in the minority, accuracy value cannot represent the performance of the prediction model.
- The aim of their designed predictor is to detect true positive defects on future projects, so self-test result is not reliable.

To deal with issue 1, despite calculating prediction accuracy, they measured the confusion metrics to calculate false alarm rates, detected rates in this situation. For issue 2, instead of using the self-test method, they selected attribute subset iteratively by M*N cross-evaluation, where the dataset is divided into N buckets, and for each bucket performed M-way cross-evaluation. After comparing the performance of different classifiers, they determined to use Naive Bayes (with log-transforms) in their predictor.

Vulnerable Code Change

The Vulnerable code change is defined by Bosu et al. in paper[8]. This paper identified which characteristics that indicate which code changes are more likely to contain vulnerabilities by analyzing several open source software. There are three main aspects studied in this paper, including code attributes, code commit characteristics, and human factors. They processed mining procedure on peer code review data since peer code review data documented rich discussion between developers regards to potential vulnerabilities. In the aspect of characteristic of developers, this paper raised two hypothesis, first one is "Does an author's experience affect the likelihood of writing vulnerable code changes", second one is "In cases where an open source project is sponsored by an organization, are authors employed by that organization less likely to write vulnerable code changes than other authors?" By building the initial set of keywords. This kind of text mining method for detecting vulnerabilities can also be used in mining GitHub commit comments data. However, most developers did not develop a reliable coding habit during their work, and their commit comments are more likely to be "fix all things" which has no meaning for text mining. So we can not collect rich commit-comment database for analyzing the characteristics of vulnerable code changes in this project.

Characteristic of Applications and Vulnerabilities

Scandariato and Walden[9] designed a new model to predict a component of the application is likely to be vulnerable or not via text mining technique. They generated a vector of monogram frequency to represent each java file as independent data.

During the development of detecting vulnerabilities, researchers looked into different levels to extract features that can be used for detection from a project. Some try to relate characteristics of the software product itself with vulnerabilities, for example, studying the complexity of the source code. Another aspect is to study characteristics of the process of code development by tracking developers behavior or modification on the file.

Shin first discussed whether software complexity relates to the appearance of vulnerabilities or not in his early research[10]. By using metrics tools, they collect the following features to numerically present software complexity, including cyclomatic complexity with different measurement methods, nesting complexity, number of possible paths, number of lines and executable statements. After drawing a conclusion that a vulnerable file seems to be more complex than a neutral file in this early work, he started another study taking both aspects into account including internal and external characteristics in his paper[11]. They made three hypotheses for establishing complexity metrics: First, vulnerable files are more complex than neutral ones. Second, vulnerable files have a higher unit complexity than neutral ones. Third, vulnerable files have a higher coupling than neutral ones. Forth, vulnerable files have a lower comment density than neutral ones. In addition, they also made hypotheses on code churn. For example, a higher code churn results in vulnerable files, frequent check-ins makes vulnerable files, and vulnerable files have more lines of code that have been changed than neutral files. As have mentioned before, developer activity was also considered, they made following hypotheses for developer network: vulnerable files are more likely to have been worked on by non-central developers, vulnerable files are more likely to be changed by multiple, separate developer clusters than neutral files.

Based on these hypotheses above, they collect corresponding metrics from the project, and then do an evaluation on each metrics categorizes by using discriminant analysis. It turns out that organizations can use complexity and developer activity metrics to proactively improve software security.

Predicting Vulnerabilities by Measuring Code Churn

Code churn is a measure of modifications happened during development an application or a software, and it can be extracted from development log files, which are recorded by a version control system. Despite recording basic information about one change, such as responsible developer, modification time, a version control system is able to show the difference between previous versions and later versions. And these information are all the basis of code churn measurement.

To study further about the relationship between vulnerabilities and code churn metrics, it is also meaningful to look into those researches which is relating software defect density with code churn. Tracing back to 2005, Nagappan et al.[12] used statistical regression models to prove that although absolute measures of code churn have weak relationship with defect density, they calculate several relative measures of code churn is a good predictor of software defect density.

Predicting Vulnerabilities with Exploring Developer Network

Shin is not the only one that takes developers network into account while studying software vulnerabilities. Meneely[13] explored the relationship between software failure and developer network. In order to quantitative variations on the centrality and connectivity of a developer network, they capture the number of developers working on same source code file, and calculate the betweenness of node v based on equation below, where $\theta_{st}(v)$ is the number of geodesic paths from developer s to t passing through node v and θ_{st} is the total number of paths from s to t. Pointing out that a developer with high betweenness means being central in the network.

$$B(v) = \sum \frac{\theta_{st}(v)}{\theta_{st}} \tag{2-1}$$

[14] [7] After building a developer network based on code churn information, then summing up metrics of developers who contribute to one file to present file-based metrics. Comparisons were made to examine the performance of the promoted model in practice, including comparing the source-lines-of-code basic model with promoted model and comparing model containing code churn metrics only or network metrics only with this promoted model. According to their study, a degree was positively correlated and closeness was negatively correlated with software failure. Their conclusion motivates this research to take developer network of a application into consideration.

2-4-2 Program Statement Level Analysis

Another type of study object is the program statements. Unlike file level study, statement study on predicting vulnerabilities considers more about input and output data.

Detection Model Based on Taint-based Analysis

In spite of having an overview on the whole context file, researchers also looked deeper into source code by studying each statement inside to precisely detect the location of existing vulnerability. Shar first started his work [15] on proposing a method to detect and automatically remove XSS vulnerability. Their job can be mainly divided into two different phases. To detect potential XSS vulnerability, they used taint-based analysis technique which is a typical static analysis on detecting vulnerabilities. In the second phase, they designed two steps to remove XSS vulnerability. Firstly, identifying the statements referenced by untrusted data in an HTML output statement can be escaped without influencing intended HTML outputs and security aspects. Then extracting the HTML document structure surrounding each untrusted data from the source code and using pattern matching to identify the HTML context. Secondly, generates secure code structures using ESAPI's escaping APIs, which created by OWASP[16], as replacements for the original code, and this process is fully automatic.

However, taint-based static analysis, which is based on predefined regulations to determine whether the source code is vulnerable or not, could cause a large amount of false positive results during the practical use.

Detection Model Based on Hybrid Analysis

Philipp's paper[17] introduced tainting-based dynamic method to detect malicious part. Data taint starts from marking untrusted data as malicious data, then propagate through the program and track the propagation. By following this procedure, traditional server-side protection mechanisms like [18] [19] could successfully prevent tainted malicious data from being used. Unlike traditional tainting-based approaches applied on the server side, Philipp proposed an approach on the client side by modifying Firefox Web browser. Meanwhile, he pointed out that dynamic techniques cannot be used for detection of all kinds of control dependencies, so their "dynamic" tainting method is actually a mixture of static and dynamic techniques in order to achieve a full protection against XSS attacks.

So based on Balzarotti's paper[20], Shar promoted a new prediction model for preventing injection attack by introducing hybrid analysis to collect features from the source code. In the first place, Shar focused on studying two vulnerabilities including SQL injection and cross-site scripting in shar2013mining, and using a data dependence graph to present a web program. Each node in the dependency graph that they were studying during feature selection and classification was "sink". A sink is a program statement that would interact with a database or web client. According to their static and dynamic analysis, they extract 22 attributes from the data dependence graph and use them as input to train and test classifiers. To collect the static analysis attributes for each sink, Shar used an open source analysis tool called Pixy [21], which is specially designed for analyzing PHP language.
Afterwards, Shar wrote another paper[22] that leverage control dependency information instead of data dependency one to make some progress on this prediction model. Despite, more efforts on classifying method was also made by introducing a semi-supervised classifier into

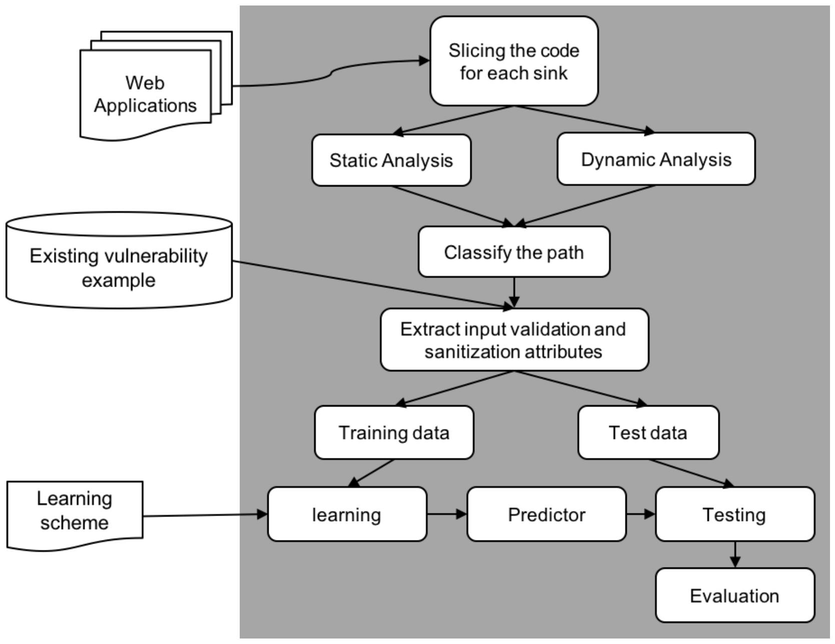


Figure 2-2: program statement level vulnerability prediction framework proposed in[22]

this prediction model, in order to deal with lacking labeled vulnerability information.

Both two works are based on a framework in Figure 4-1, which includes two main parts, including generating feature vectors for each sink and training classifiers to filter vulnerable sinks from neutral sinks. They concluded 22 attributes based on static and dynamic analysis both in [22], and all the attributes that they used are numeric. Table **??** shows the attributes that are extracted by using static analysis only. Point out that these attributes all have clear definitions concerning security requirements consensus or are directly associated with known vulnerability issues, they could be predefined easily and collecting these metrics can be made statically.

Despite standard security functions, some operations might involve complex string manipulations, simple static analysis cannot help to classify the attributes, therefore dynamic analysis is needed.

Article	Approach	Focusing area	Limitation
Shar and Tan 2011[15]	Taint-based Static Analysis (Java)	Detection of stored and reflected XSS vulnerability	high false positive rate in results by using taint-based static anaylsis, might miss some vulnerabilities since the method do not track information flow across web pages.
Shar and Tan 2013[23]	Hybrid Analysis (PHP) + cluster	Detection of SQL injection and XSS vulnerability	Not accurate as full static or dynamic analysis
Shar and Tan 2015[22]	Hybrid Analysis (PHP) + semi-supervised	Detection of SQL injection and XSS vulnerability	static and dynamic analysis results are acheived by using Pixy[21]

Table 2-1: Comparing different detection model

Chapter 3

DATASET AND FEATURES

This project focuses on studying three main aspects of the GitHub repository, including source code complexity, developer network structure and developer behaviors. This Chapter will be separated into the following section: First, it would focus on describing what kind of resources I can get access to extract features for detecting vulnerabilities, and explaining why I use them for my project. Then, the definitions of all features selected to generate a feature vector are shown in the second section, also briefly introduce the methods of calculating these metrics. Next, noticing values from different platforms are differ from each other, I introduced normalization process. At last, using visualization method to present the feature values and generalize useful suggestions to developer team.

3-1 DATA RESOURCE

This project focus on three main aspects of the GitHub repository and the data was provided by eBay company, including source code complexity, developer network structure and developer behaviors during constructing platforms. A raw dataset consists of all JavaScript source code files and whole commit history files from GitHub.

- Source code files
  This project is a code file level study, we treated each file as a unit. Every feature we used here is for constructing a feature vector for each file. In the previous study on detecting vulnerabilities in applications, researchers studied the relationship between source code complexity and vulnerabilities, based on their result, we designed to extract complexity metrics from source code file to study further.
- GitHub commit history
  The human factor, which is usually being neglected during designing a defect detection model, is actually an important aspect to look further. One way to learn about developers habits is to conduct a survey on developer group and gathering the information. But limited valid data can be retrieved from this survey, and by this way, updating

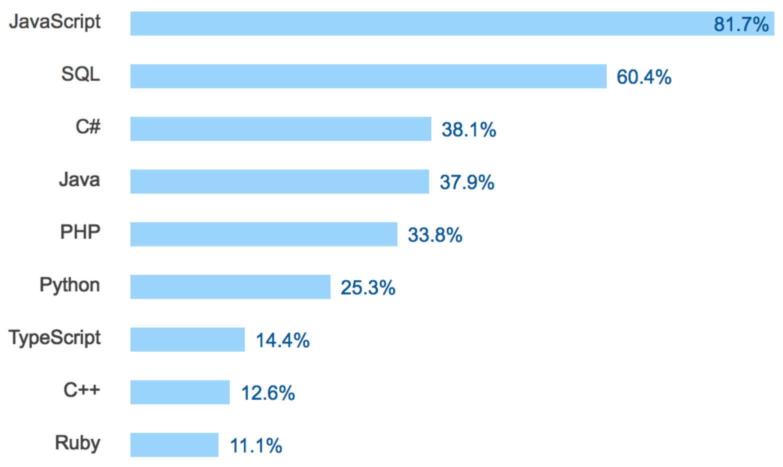


Figure 3-1: results of web developer's most popular programming language

detection model along with the changes on developer group setting can only be done by conducting newer surveys, which is not efficient. In order to learn extendable information from developers, it is a wise idea to track their developing behaviors.

3-1-1 Source Code File

According to the annual developer survey conducted by Stack Overflow in 2017, JavaScript is the most popular programming languages among web developers, which is also shown in Figure 3-1. Why is JavaScript widely used for web development? Firstly, JavaScript is an object-oriented language with prototype inheritance, which allows objects to inherit properties from each other directly. Secondly, due to the success of Node.js and its large open-source library "npm". Node.js is initially designed for data-intensive real-time applications and keeps to be lightweight and efficient, and it also enables it to execute JavaScript code on the server side. Thirdly, JavaScript is available in many web browsers.

Some vulnerabilities are related to an unsecured design of web application structure, but the features used in this paper are more related with the quality of coding. So this paper only focus on those risks caused by unsecured coding.
Before collecting metrics from source code files, I put some efforts into studying the web application coding file structure. As is shown in Figure3-2, Java or Scala file is developed to collect requested data or store processed data. However, before sending these collected data to web browsers, it is necessary to do pre-processing on raw data. Using unsanitized data is possible to exploit a weakness to attackers, for example, it could lead to cross-site scripting or SQL injection etc. And these pre-processing data methods are settled in JavaServer Pages or imported to JavaScript files from manually designed packages.

There is a trend that Java or Scala for the server side data processing will be replaced by the Node.js framework in commercial platforms. Current and future code file structure are shown in Figure 3-2 and 3-3. In this project, since the Node.js framework was recently introduced to frontend development department, a relatively small number of developer GitHub commit history for Node.js based frontend file package can be used to study further. Current structure in Figure 3-2 has been used for more than 7 years, and adequate commit history record is available in GitHub repository.

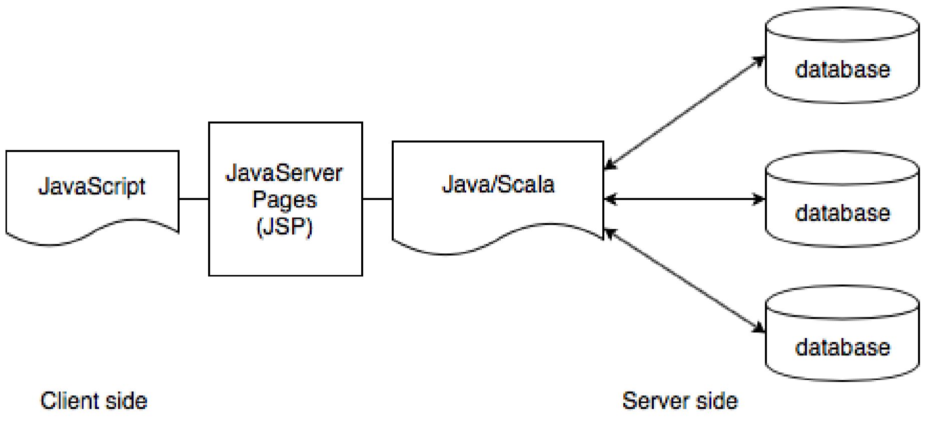


Figure 3-2: current frontend code file structure

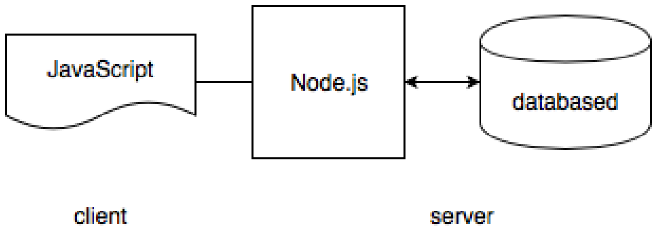


Figure 3-3: future frontend code file structure

3-1-2 GitHub Repository

With the rapid increase in the popularity of decentralized source code management(DSCM), GitHub becomes one of the most important resources of software artifacts on the Internet[24]. A large number of works on mining GitHub's event logs have proven the rich informativeness that GitHub owns. In essence, GitHub is an online version control system. As mentioned in Chapter 2, previous researches have already related software defect with code churn measures, and proven the close relations between them. This paper was inspired by the idea of associating web application vulnerabilities with code churn measures. Moreover, the previous study put efforts on discovering the relationship between developer network characteristics and software failures. On the foundation of their research, I introduce social network analysis into my research. In order to have basic information about developers, I pay attention to the information provided by a version control system. It is easy to build a developer network based on GitHub repository data.
GitHub is a popular social coding site that uses Git as its distributed revision control and source code management system.

3-2 FEATURE DEFINITION

As briefly introducing some researches of predicting software fails and web application vulnerabilities, various of aspects can be selected to study further. Considering all accessible resources and aim of this research, this research will look at three aspects: code complexity, developer behavior, and developer network. For preparing the feature vector of this project, a large number of calculations is needed. The target of feature engineering procedure in this study is to define some distinguished features to identify files from different classes and format these feature vectors as the input of the selected machine learning algorithm.

Code Complexity Metrics Code quality assurance is the foundation of preventing vulnerabilities. Despite a limited number of vulnerabilities are caused by inappropriate framework design, most of them are code-level related. In this project, complexity metrics are collected to represent objective code quality by applying static analysis. Halstead[25] and McCabe[26]'s metrics are collected. Definitions of these metrics are shown in the following:

- path: the file path to be used as an index
- sloc: physical line of codes, which is the number of lines in a module or function
- cyclomatic: the number of cycles in the program flow control graph.
- cyclomatic density: a percentage of cycles in the logical lines of code
- operator: the total number of operators
- operand: the number of distinct operands
- vocabulary: the sum of distinct operators and operands
- length: the sum of appearance times of operators and operands
- difficulty: $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$ where η_1 is number of distinct operators, η_2 is number of distinct operands, N_2 is total appearance number of operands.
- maintainability: derived from the logical lines of code, the cyclomatic complexity, and the Halstead effort.

Developer Behavior Metrics In order to quantify the developer's behavior during development, I used the information from GitHub commits. Each commit includes basic information about who edited the code, when it happened, which line is modified. By crawling all GitHub commit history per each platform, we are able to compute some basic metrics and some derived metrics that are showed below:

- num_churn: the total number of modification times on source code file
- duration: the number of days since each code file was created till last edited date
- frequency: the average number of commits within a day

- commit_du: the average duration between two commits
- num_dailychurn: count the number of days that a code file changes were committed several times a day
- num_dev: the number of developers who have done contribution to this file
- work_dev: the number of developers who are still making contributions to this file
- leave_dev: the number of developers who stopped contributing to this file.

Developer Network Metrics The measurement of features for each file is based on developer network measures, so it is necessary to create a developer network first. In this undirected developer network, each node represents a contributed developer and we add edges between two nodes when both of them do contribution to the same code files. To quantify developer network, graph theory is introduced to this project.

- degree: the sum/average of each developer's degree
- closeness: the sum/average of each developer's closeness centrality
- betweenness: the sum/average of each developer's betweenness

3-3 FEATURE COLLECTION

3-3-1 Code Complexity Metrics

Since each file is characterized by looking into three different aspects, the method of collecting different metrics for each one varies from each other. In this project, in order to identify these files, the unique index is their file path. As described before, we only focus on collecting attributes from JavaScript files in this project. For analyzing JavaScript files, Phil Booth implemented an open source tool called "PLATO"[27] on the basis of Halstead's and McCabe's theory for generating code complexity statistical report. "PLATO" is able to find all JavaScript file existed in the development package automatically, and it returns a JSON file report that contains all details of each JavaScript file.

3-3-2 Developer Metrics

In order to compute code churn measures and characteristics of the developer network, we should have data that records what developers have done during the development. In order to crawl GitHub commitments history from the GitHub repository, this research uses *PyDriller*[28], which eases the commitment information extraction procedure. *PyDriller* is able to write a detailed record. According to the author's description, there are two domain objects showing below:

- **Commit** This domain object is designed to store all the information related to a commit:the hash(which is a unique code for each commit), the committer (who have the right to confirm a commit), the author (who does the modifications), the message (comments left by author), the authored and committed dates (useful for computing some features like frequency and duration), a list of its parents' hashes (a merge commit has two parents), and the list of modified files (create a new domain 'Modification' to store the information about each modified file).
- **Modification** modification object is settled to carry information about file modification in a commit history, and this object has several fields:
  - **Filename**: name of the modified file
  - **OldPath**: path of the file before modification
  - **NewPath**: path of the file after modification
  - **Change Type**: including *ADD, RENAMED, MODIFY, DELETE*
  - **Difference**: code line difference between versions, and shown in *Git* way.
  - **Added**: the number of added code lines
  - **Removed**: the number of removed code lines

Developer Behavior Metrics

Before computing developer behaviors and developer network metrics, the first step is doing initial cleaning on these commit records, and it is an important step regarding modification type. According to the definition, there are four main kinds of different modifications. Among them, only *ADD, MODIFY, DELETE* operations are valid modifications to be considered during calculating developer behavior metrics. It is because that *RENAMED* operation has no impact on the code quality and no relationship with the existence of web application vulnerabilities.

Not all information provided by Git will be used in this research, developer behavior features are only related with author identity and time. Table 3-1 shows all defined developer behavior metrics with their corresponding Git commit information details, in other words, these metrics are measured by considering this information. The computation for these features are directly derived from Git commit information, but we have to make sure that the definition of these features is in accord to actual situation of code files. The challenging part of collecting developer behavior related metrics is to determine the definitions of these features. It is crucial to define *duration* because *frequency* and *commit_du* both depend on it. I generate two reasonable definitions based on the understanding of software development life circle, one is measuring days since a file was created till it was last edited, another one is calculating days of existence. In order to determine which one to use in this research, I visualize the distribution of *duration* with these two definitions separately, details and proven are shown in section 3-6.

Despite *duration*, *work_dev* and *leave_dev* also have ambiguous meanings in actual use. These two features are initially designed to categorize developer with contributions. Most Internet

Feature	**Related Git Commit Info**
num_churn	NewPath, Change Type
duration	NewPath, commited date, Change Type
fequency	NewPath, commited date, Change Type
commit_du	NewPath, commited date, Change Type
num_dailychurn	NewPath, commited date, Change Type
num_dev	NewPath, author
work_dev	NewPath, commited date, author
leave_dev	NewPath, commited date, author

Table 3-1: Developer behavior features with corresponding Git commit information

companies have multiple developer groups working on diverse projects, it is common to have a high turnover of technical personnel within them. Instead of collecting active developer name list from each developer group manually, this research settles an automatic method to distinguish working developer and left developer by seeing their commitments history. If a developer stop contributing to a file over months, it is possible that this developer was moved to other teams or had already left the company. So, we can categorize these inactive developers into left developer regarding each code file. On the contrary, the developer who has done contribution to this file within months will be categorized into working developer. How to determine the boundary between the working developer and left developer is discussed in section 3-6.

Developer Network Metrics

Unlike measuring developer behavior metrics, collecting network metrics for each file to quantify the developer's experience and group work quality are based on network analysis. In this part, we will introduce graph theory and social network analysis.

In network analysis, each vertex is called nodes in a network graph, and every edge between two nodes is named as a connection. Despite these two essential elements, a sequence of non-repeating, adjacent nodes is a path, and the shortest path between two nodes is called a geodesic path. From the perspective of social network analysis, a geodesic path is the "social distance" from one node to another in social network analysis.

In terms of characteristics of code files, it is unrealistic to do social network analysis on a code file network directly, so procedure of collecting network metrics per each file can be divided into two phases:

- **Constructing Developer Network** In this undirected network, each vertex node represents an individual developer, including active and inactive developers. The connection between two developers represents that they have worked on the same code file before. Moreover, the weight of the edge between nodes is accumulated by the times they work together. An example network graph is shown in Figure 3-4, in this figure, I colored edge with large weight (more than 3 times) in *Red* and other edge are colored by *Orange*.

- **Collecting Developer Network Metrics** By introducing social network analysis into this research, we can quantitatively measure the structure of a developer network. According to the previous study, metrics that measure direct connections between vertex nodes are **connectivity** metrics, and other metrics that measure how each node are indirectly connected to the rest of nodes in the network are **centrality** metrics. The target of applying graph theory in this research is to interpret network structure in a statistical way. Among all network metrics, this research will use *degree, closeness* and *betweenness* to study further. All these 3 metrics all have intuitive meaning in a developer network. *Degree* can represent how experienced a developer is by calculating how many developers and files he/she had worked with. *Closeness* is derived from shortest path measures, which can be used to show the way of each developer working with others. *Betweenness* is a metrics can evaluate whether the developer is usually the center of the team or not. Defined measurement equations for *closeness* and *betweenness* are shown below, where $\sigma_{st}(v)$ is the number of geodesic paths from node s to t going through node v, σ_{st} is the total number of geodesic paths from node s to t for *betweenness*, and $|V(G, v)|$ is the number of reachable nodes in the graph of node v, $d_G(v, t)$ is the number of edges from node v to t.

$$Closeness(v) = \sum_{s,t,v \in G} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3-1}$$

$$Betweenness(v) = (\frac{1}{|V(G, v)|}) \sum_{v,t \in G} d_G(v, t) \tag{3-2}$$

- **Generating Network Metrics for each File** After quantifying the experience of developers and the way they work with others, the next step is to use these collected developer network metrics to measure each file. From GitHub log file, I can collect a list of contribution developers for each file. Based on previous steps, we have developer-based network metrics already. When determining file-based network metrics, I use the average and sum of developer-based metrics.

3-4 LABEL DATASET

The web application is facing with various types of vulnerabilities, but in general, they can be categorized into two main kinds based on the causes of them, including design flaw and implementation bug. Obviously, most design flaws are hard to detect by only analyzing individual source code files. In this research, only implementation bugs caused vulnerabilities are considered.

When doing classification on these data sample by using machine learning algorithms, we do not subdivide web application vulnerabilities into specific kinds of vulnerabilities. It means that dataset in this research only is binary, in other words, a file can only be vulnerable or normal. For labelling collected files, we have two resources. One is vulnerability reports provided by third party security testing company. Another one is reports generated by automatic security check tools. The major difference between these two kind of resources

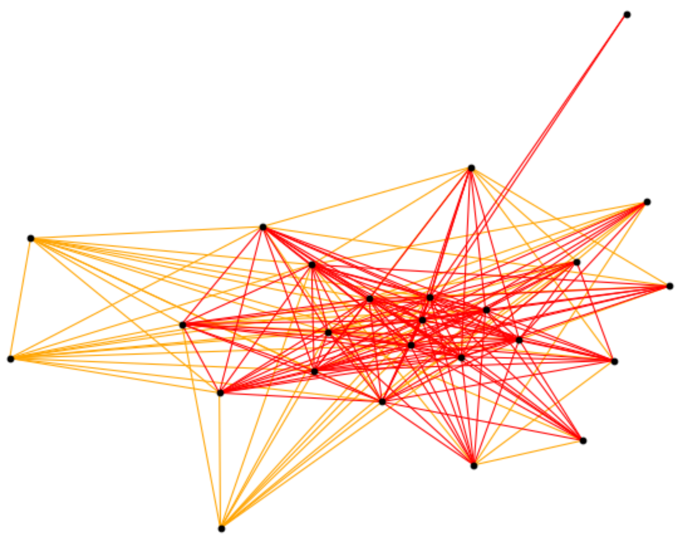


Figure 3-4: An Developer Network Example Presented by Graph

package	code file number	vulnerable file number
pck1	156	4
pck2	452	2
pck3	146	7

Table 3-2: Package information: Total number of JavaScript files and number of vulnerable files

is that testing company has no access to source code files, so they can only point out which website is vulnerable. While security check tools have full access to all data files, the detection result of these tools are accurate to the source code file. I show the number of vulnerable files along with number of all existing code files separately for each package in Table 3-2. In average, there is only one vulnerable file within 58 files. So when designing the prediction model for this research, I need to consider how to deal with imbalanced data in priority.

3-5 NORMALIZATION

In this research, data samples are collected from different development packages and platforms. For improving the performance of classifiers, it is essential to have data set in value consistence. To see whether values collected from different packages are consistent, I visualize the distribution of each feature.

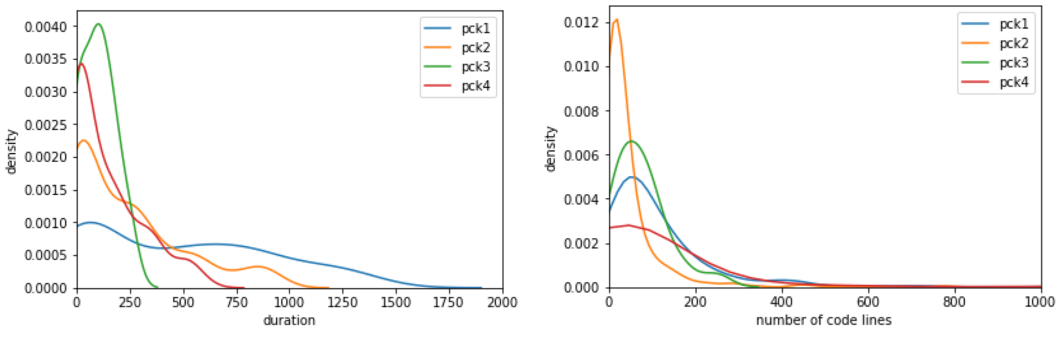
The distribution of feature duration and line of code (sloc) is shown in Figure 3-5 as examples.

The *duration* metric represents the existence days of a file, and the value could be influenced by several factors, for example when the platform was established, how often the framework version was updated. As a result, the distribution of *duration* varies from development package to package. The *duration* of package 3 ranges from 0 to 350, however that of package 1 distributed from 0 to around 2000. The distribution of *sloc* metric is similar to *duration* since the distributions of metric values are quite different between these packages.

Due to the inconsistency within different packages that I used, the normalization process is necessary before learning patterns from these collected data. In this research, I apply "min-max" normalization strategy which transforms x to y following the equation below:

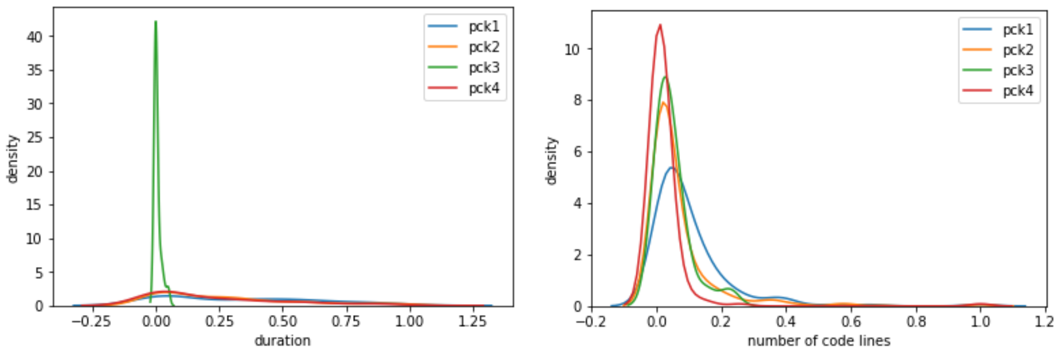
$$y = \frac{x - min}{max - min} \tag{3-3}$$

The density curve of *duration* and *sloc* are plotted in Figure 3-6. Apparently, the dataset after normalization process is consistent between different packages, and the value ranges from 0 to 1 without influencing the original value distribution.



(a) *duration* (b) *line of code*

Figure 3-5: Feature value distribution before Normalizing



(a) *duration* (b) *line of code*

Figure 3-6: Feature value distribution after Normalizing

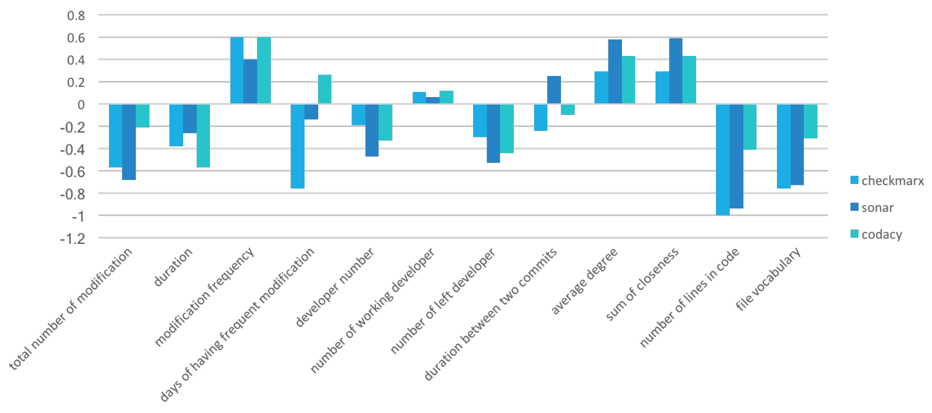


Figure 3-7: Difference feature value between vulnerable and normal files

Files are labelled by using results of different detection tools

3-6 DATA VISUALIZATION

In order to better understand the data I collected in this research before carrying out further pre-processing procedure on the dataset, data visualization is helpful for concluding initial results. In this initial analysis, I select two prospects to study further.

3-6-1 Feature Value Differential

Here, I use the dataset before normalization procedure to visualize the difference, the major problem is that feature values varies greatly from each other, which means the difference value can range from single digit to thousands. For showing difference value of various metrics in the same figure, an equation is defined in 3-4, where *Average* is the average number of all metric value.

$$Diff = \frac{Avg(Normal) - Avg(Vulnerable)}{Average} \tag{3-4}$$

The difference value of each metrics can be influenced by which files are included in Normal data samples and Vulnerable samples separately. Two factors can determine what files are included in each sample, including using reports generated by different vulnerability detection tools to label source code files and files belong to different platforms. When visualizing the difference, I utilize control variate method. In Figure 3-7, all files are from the same platform, but labels of these files are determined by results of different detection tools. And in Figure 3-8, files from different platforms are labelled based on *Checkmarx* result.

A feature, which is worth to be study further, is always with positive(negative) differential value, no matter how the influential factors are changed. As is shown in the figure, days of having frequent modification (num_dailychurn) and duration between two commits (commit_du) might be greatly influenced by using different detection tools result and also using files from different platforms. In conclusion, prediction results that is based on these two features might be unreliable.

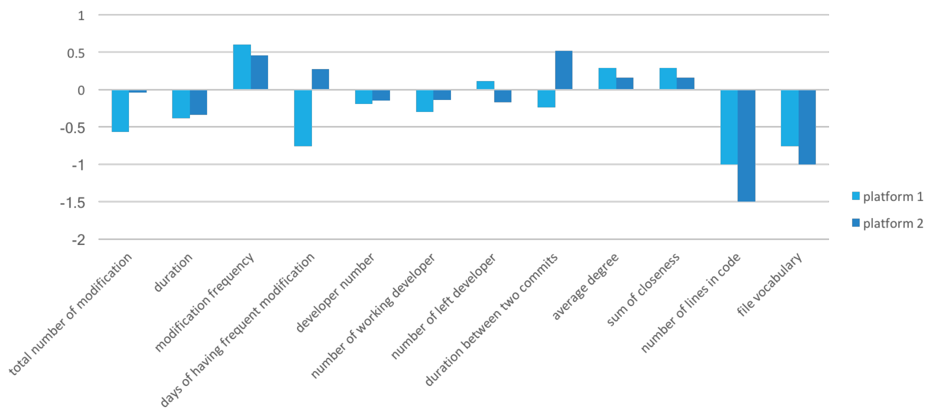


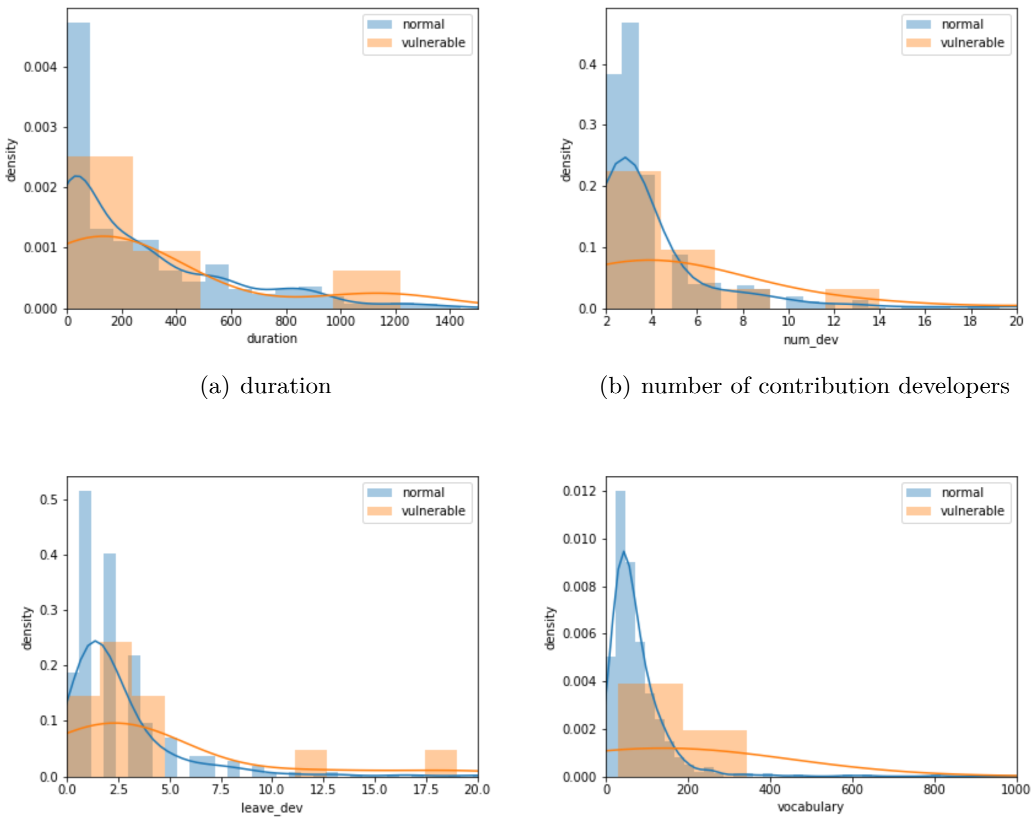
Figure 3-8: Difference feature value between vulnerable and normal files

Files belongs to different platforms

3-6-2 Feature Value Distribution

In this section, by visualizing the distribution of normal and vulnerable files' feature values separately, some initial results can be derived from comparing the data from two classes. Figure 3-9 shows the distribution of four features, including *duration, num_dev, leave_dev* and *vocabulary. Duration* measures the time that a file exists, the value distribution indicates that most of the normal files have shorter existence time. Several vulnerable files have existed for a long period of time, which make it different from most normal files. *num_dev* is a metric to count how many developers have contributed to a file. More people can produce more implementation bugs, while more people are more likely to discover the bugs and fix them. Various factors can influence the relation between developer and vulnerability. As can be seen from the figure, most normal files have contribution from only 2 to 4 developers. Also the distribution between 2 to 8 of vulnerable files is similar to normal files, so when assigning works to developer, 2 to 4 developer in a team is enough for developing a file. On noticing the frequent personnel changes, attentions are also paid to analyse whether number of left people are related with web application vulnerabilities. The figure shows that vulnerable files are more likely to have more people left the group. The last figure is related with code complexity metrics. When *vocabulary* is larger, the file is more complex. As can be seen from the figure, vulnerable files are more complex than normal files.

Another important aspects in this research is introducing network theory into evaluating developer group work. In Figure 3-10, I first display the relationship between closeness and degree derived from network structure. It is obvious that degree value increases along with the increase of closeness value. We can also interpret the relationship as that experienced developer group tends to work more closely to each other. The other two plots show the distributions of these two file-based network metrics. *closeness* density curve shows that peak point of normal file density curve is with larger *avg_closeness* value than that of vulnerable files, which indicates that a development team with close collaboration is more successful in maintaining files to be secured. Different from the density curve of *avg_closeness*, the majority of normal files are developed by group of developers having low degree value. However, it is also obvious that none of the vulnerable file is developed by a group of rather experienced developers.



(a) duration (b) number of contribution developers

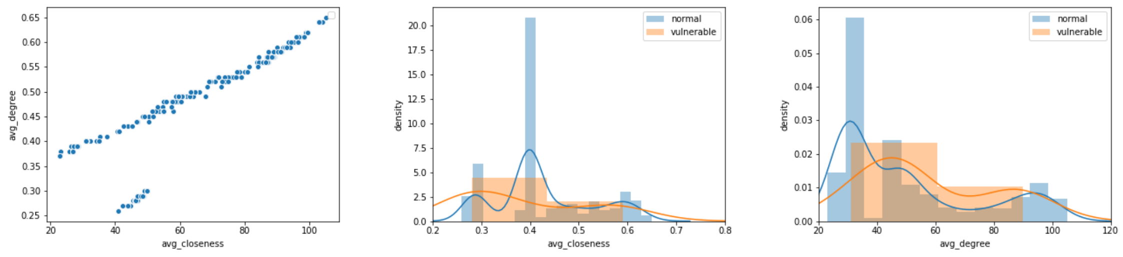
(c) number of left developers (d) vocabulary

Figure 3-9: Feature value distribution of normal and vulnerable files

3-7 Discussion

In conclusion, code complexity metrics can be achieved from inspecting source code files themselves, and the static metrics related with developer behavior can be extracted from GitHub log files directly. In order to measure the metrics of developer team, it is necessary to introduce social network theory into this research. Despite collecting feature values, labeling each file is also the foundation of supervised learning. Since this research uses the results of checkmarx security report, file labels might have false positive results. However, this issue could be solved in the future. To analyze collected data, I inspect the facts by visualizing these values by using different kinds of visualization methods.

According to the information that can be learned from these visualization figures in this Chapter, several suggestions are proposed for developers or development group manager to consider about. To clarify these suggestions clearly, I also divided them into different



(a) Relationship between closeness and degree (b) average of closeness (c) average of degree

Figure 3-10: File-based network metrics relation and distribution

categories.

Source code files

- Avoid creating complex source code files. The prevention methods include dividing a file into small parts, each function is implemented in an individual file.
- Accelerate the pace on updating the package framework. From the *duration* metrics, we can know that some files have existed for a long time. If the framework of packages are frequently updated, there would be not that much old files which exist over 3 years in a development package.

Developer team

- For each file, 2 working developers is enough to maintain the security problem. By calculating how many developers are still contributing to each file, it turns out the maintainence work do not require a large group of developers. Assigning more people is a waste of resource and also has risk on creating more security issues.
- Prevent developers from leaving the developer team. According to the study, it shows that number of left developers is related to the appearance of vulnerabilities.
- Motivate developers to work closely with each other. In order to enhance the collaboration between developers, it is also important to have experienced developers in the team.

For supporting code complexity and developer related metrics are important for building vulnerability prediction model, the next chapter would first introduce the prediction model framework for this research and doing different aspects of analysis on the prediction result to answer the research questions raised in the beginning of this thesis.

Chapter 4

EXPERIMENTS

This Chapter includes three parts to introduce the experiments in this research. The first section describes an overview of the structure of my designed framework. Then, I will explain the details of each procedure in this experiment, including the settlement of training and testing parts, along with introducing data pre-processing procedure and evaluation metrics or methods that would be used to compare the performance of classifiers. Finally, to understand the testing results generated by this prediction model, different prospects are selected to study further.

4-1 Framework

In Chapter 3, I describe the way of cleaning the dataset. These collected data are the input of my designed prediction model. In Figure 4-1, I present the framework of this prediction model. Follow the diagram; the whole process starts with cleaning raw data and collecting features from source code files and GitHub commit history. Because of using different aspects to inspect the raw data, the collected features can be categorized into three kinds. Afterward, I do feature normalization on feature values, and this procedure should be done for different platform file separately. Then, the whole dataset would be divided into two parts: the training set and testing set. During the training process, the main purpose is to achieve a best-performed classifier that can be used to generate the final result of this model in the testing phase. Finally, analyzing the obtained result to the reasoning why it is a good model or not regarding system setting, and concluding instructive information to give suggestions to platform developers.

4-2 Training

The whole procedure follows the procedures below:

- Feature extraction by Principle Component Analysis(PCA)

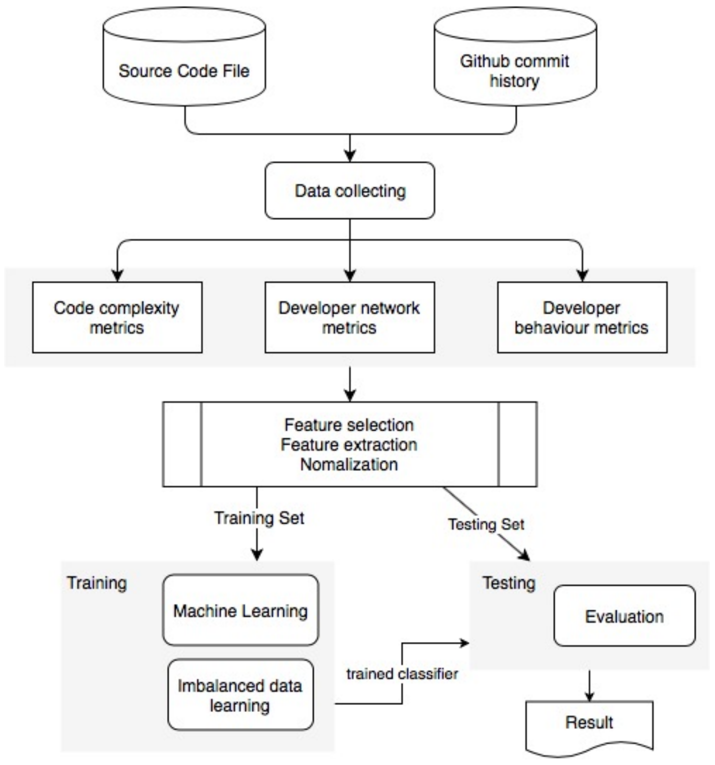


Figure 4-1: Vulnerability detection model using machine learning scheme

- Generating training and testing dataset
- Data resampling on training data
- Classifier parameters settlement
- Classifiers assessment

4-2-1 Imbalanced Learning Criteria

Classic machine learning researches are aiming at improving prediction accuracy. For imbalanced learning, accuracy is not the only evaluating criteria to compare the performance of different classifiers. True Positive Rate (TPR) and False Positive Rate (FPR) are often used to evaluate the performance of different classifiers. G-mean and AUC value are often used to measure the performance of predictor on balancing two classes for having a more comprehensive evaluation of predictors in the imbalanced data. G-mean is defined in Equation 4-1. A classifier with good performance should achieve high accuracy for positive and negative classes at the same time, so a good classifier should have higher G-mean value. AUC value measures the area under the ROC curve, and the ROC curve presents the relative trade-off between FPR and TPR. A ROC curve shows the performance of a classifier across all possible decision thresholds. A better classifier should have a higher AUC value. Despite these two criteria, *balance* metrics is also used to evaluate the performance of classifiers. As we known, The ideal point for ROC curve is the point with $FPR = 0$ and $TPR = 1$, and *balance* measures the Euclidean distance from the real (FPR, TPR) point to (0, 1). The higher *balance* is, the better a classifier is.

$$G - mean = \sqrt{recall(1 - FPR)}; balance = 1 - \frac{\sqrt{(0 - FPR)^2 + (1 - recall)^2}}{\sqrt{2}} \tag{4-1}$$

4-2-2 Parameter Tuning

Before comparing the performance of using different classifiers to predict vulnerabilities, it is essential to ensure. To help to improve the result, I introduce an oversampling method to deal with the imbalanced issue and PCA method to avoid over-fitting. The process of tuning parameter for these two methods are shown below.

Oversampling Method

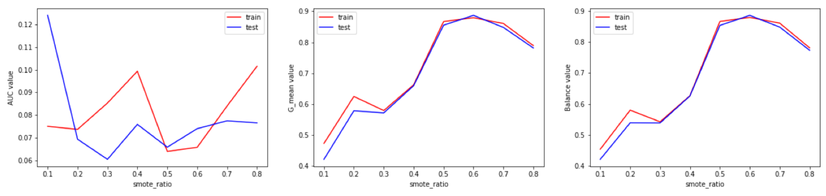
SMOTE method is widely used in the imbalanced learning area. This special oversampling method is designed to simulate the distribution of data samples existed by creating synthetic new data. According to previous researches, *smote_ratio*, which is a parameter that can control the percentage of data to be resampled. For finding which ratio value for SMOTE method can improve the performance of the prediction model. Concretely, I apply a 5-fold cross-validation method to this research. To avoid the random situation, each run would repeat five times. In total, a classifier along with using the same oversampling ratio will have 5*5 results, which means every point in Figure 4-2 is the average number of 25 results. Moreover, in order to compare the performance, we consider all three metrics mentioned in the previous section for both training and testing data. Details in figures can tell the increasing of G-mean and balance along with increasing SMOTE ratio value. Meanwhile AUC value fluctuates by using different SMOTE ratio. So the decision on SMOTE ratio depends on finding the highest AUC value. In summary, for Naive Bayes, I choose to use 0.4 as the ratio; For the Random forest, 0.4 is also the most suitable choice; For Decision tree, 0.5 is more suitable; For logistic discriminant analysis, 0.5 is also a reasonable value.

Principle Component Analysis

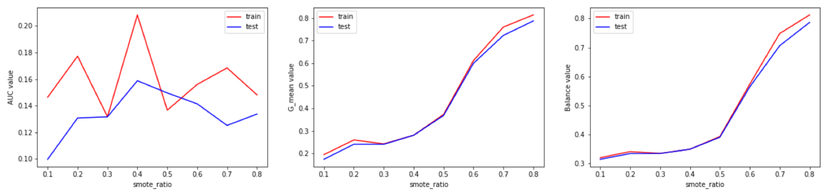
In total, I collect 26 features for about 1000 files from three different aspects of platform development. However, most classifiers are not good at dealing with the high dimensional dataset, so we need to do feature selection or extraction to reduce dimensions of the dataset. Feature extraction is a procedure of transforming raw data into new features space which is suitable for modeling. Among all feature extraction methods, PCA is one of the most popular method for data science research. Generally, PCA compute the eigenvectors of a covariance matrix with the highest eigenvalues, uses those vectors to transfer the original data into a new feature space with n components. Selecting how many components to be used as input of this prediction model can influence the performance.

4-3 Imbalanced Learning Assessment

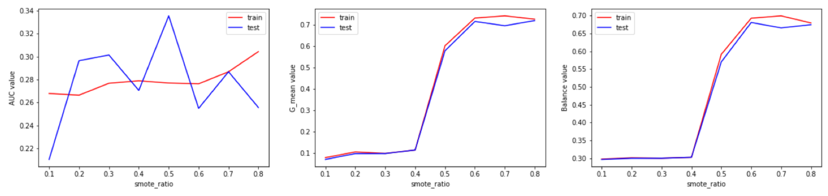
In this research, I select four supervised learning algorithms to classify collected files.



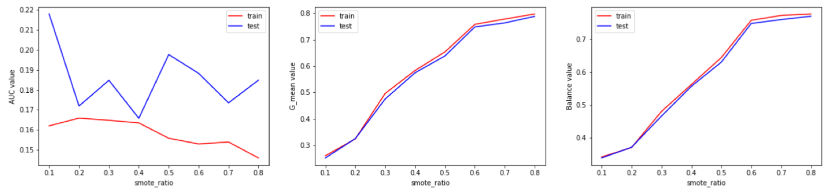
(a) **Naive Bayes**



(b) **Random Forest**

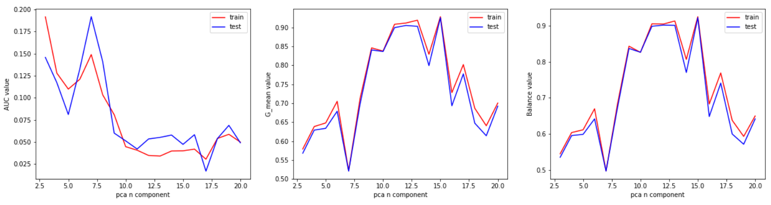


(c) **Decision Tree**

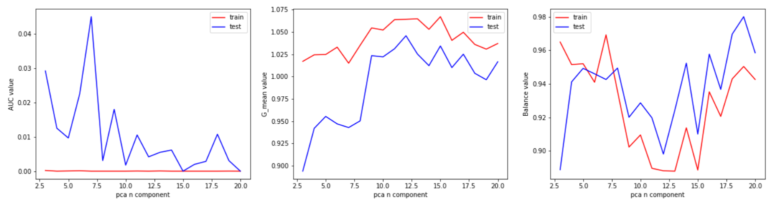


(d) **Logistic Discriminant Analysis**

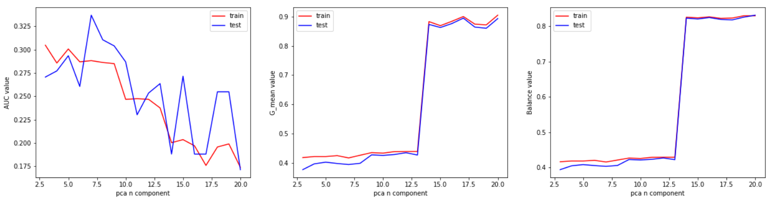
Figure 4-2: Evaluation on using Different Smote Ratio



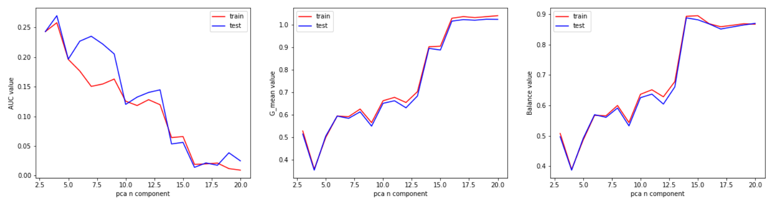
(a) assessment value of **Naive Bayes** trained by different number of components



(b) assessment value of **Random Forest** trained by different number of components



(c) assessment value of **Decision Tree** trained by different number of components



(d) assessment value of **Logistic Discriminant Analysis** trained by different number of components

Figure 4-3: Evaluation on selecting number of principle components to transform original data

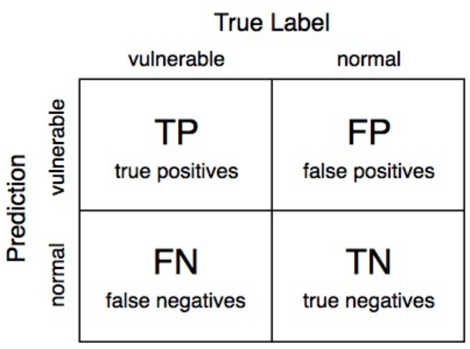
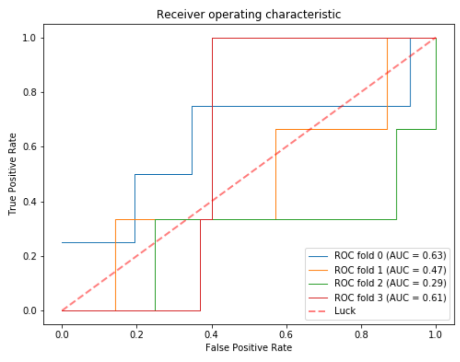
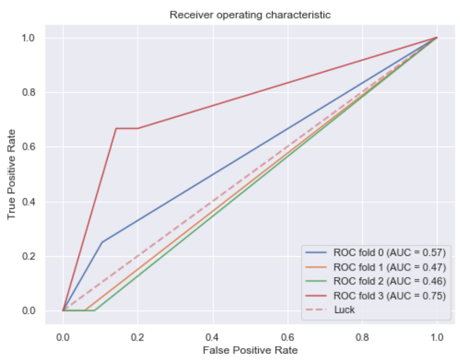


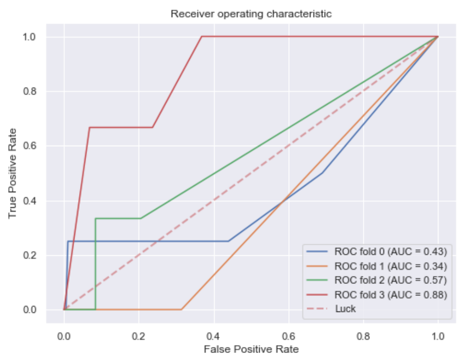
Figure 4-4: Confusion matrix for performance evaluation



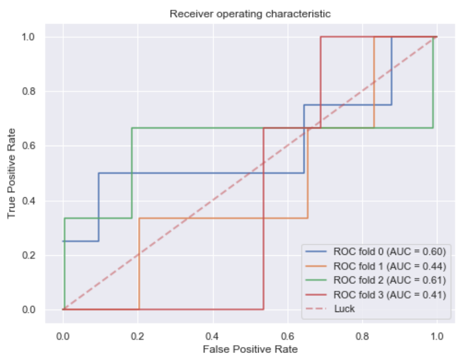
(a) **Logistic Discriminant Analysis**



(b) **Decision Tree**



(c) **Random Forest**



(d) **Naive Bayes**

Figure 4-5: Machine learning algorithms comparison based on ROC curve and AUC value

4-3-1 Receiver Operating Characteristics(ROC) Curve Analysis

In order to inspect how different machine learning algorithms performs in this designed prediction model, most researches on imbalanced data learning used ROC curve to present the relative trade-off between FPR and TPR. During this part, I apply 4-fold cross-validation on the experiment. Therefore, each fold can have an adequate number of vulnerable files. After tuning the parameters of classifiers, Figure 4-5 plots ROC curves of different supervised learning algorithms. One method to evaluate these classifiers is to compare their AUC values. Instead of computing the average number of AUC values from four folds, I compare the AUC value generated by these classifiers under using the same fold. Logistic Discriminant Analysis has the highest AUC value in Fold 0 and 1, and Random Forest ranks the first in Fold 3. However, it is difficult to compare each point in the ROC curve, so I need to introduce another comparison method to this research.

4-3-2 Probability Analysis

Despite computing metrics used in previous researches to evaluate the performance of selected machine learning algorithms, it is worthwhile inspecting which kind of data samples is more likely to be labeled as 'Vulnerable file' by our prediction model regarding their feature values. In total, this research collects 13 vulnerable files from 4 different platform development package. The idea of this comparison is similar to presenting the ROC curve, but is more straightforward to understand than ROC curve method.

Cost is defined as how many normal files are misclassified when labeling all vulnerable files correctly based on the probability estimates calculated by machine learning algorithms per each file. In Table 4-1, we list all vulnerable files with its probability of being vulnerable and probability estimation descending ranking. Ranking number directly shows the cost of labeling a vulnerable file. For example, the probability of file 635 being labeled as 'vulnerable' ranks 7, which means first seven files with higher probability estimation will be classified to the vulnerable category if file 635 is marked to be vulnerable. These labeled 'vulnerable' files would include several normal files, and these normal files are named as false positives. The ideal situation is that a classifier can be distinct from all these listed vulnerable files without producing any false positives result. From this perspective, we can compare the cost value to evaluate the performance of selected classifiers.

In order to collect probability estimation for each vulnerable file calculated by different classifiers, we use two different methods. First one is to generate a resampling dataset by using SMOTE method and then we train the classifiers on these resampling data. Then, using selected classifiers to test original dataset without resampling process and record the probability estimations. Finally, sorting source code files according to their probability estimations and recording their probability ranking place. Results are all shown in Table 4-1. Another one is introducing cross validation idea by separating the dataset into five folds; each fold is then used once as a validation set while the remaining folds make up the training set. This result is presented in Table 4-2.

We can set several perspectives to analyze which classifier gives the best performance according to these two tables. First, we compare the cost of discovering all vulnerable files. Learning from the Table 4-1, by using Logistic Regression and Naive Bayes classifier, we know that vulnerable file index 672 is the one most likely to be treated as a normal file with low probability estimation calculated by these two classifiers. It ranks the 559th and 624th place separately. However, the results of random forest vary for these two classifiers. It shows the confidence in categorizing file index 672 into vulnerable class, which probability estimation is 100 percent. Although an ideal predictor can discover all vulnerable files, this aim is always hard to be realized. So we need a trade-off between high true positive rate and low false positive rate. Considering this issue, I set another inspection on the cost of discovering 80 percent of vulnerable files. In other words, how many files will be labeled as vulnerable, if we want to have ten true positives? In this way, the random forest classifier still gives the best performance by categorizing all ten vulnerable files into positive class without producing any false positive. Learned from the table, decision tree result is different from other classifiers. Due to its settlement, the probability estimation value is binary. In order to compare decision tree's performance with other classifiers, the third perspective is designed to compute the average and variance value of file ranking result. Since a large number of normal files are classified to **Vulnerable**, the average cost of all vulnerable files estimated by decision tree is expensive, but the standard variance is comparatively low. In other words,

File	Naive Bayes		Random Forest		Decision Tree		Logistic Regression	
	Rank	Probability	Rank	Probability	Rank	Probability	Rank	Probability
672	559	0.0600	1	1.0000	437	0.1546	624	0.1266
737	123	0.4029	1	1.0000	107	0.4847	201	0.3842
738	122	0.4036	1	1.0000	107	0.4847	200	0.3849
581	288	0.1595	13	0.5083	107	0.4847	496	0.1759
710	0	0.9999	1	1.0000	437	0.1546	0	0.9400
588	495	0.0860	11	0.7000	437	0.1546	490	0.1772
734	92	0.4633	1	1.0000	107	0.4847	54	0.6286
48	284	0.1605	8	0.9000	107	0.4847	155	0.4344
722	113	0.4186	10	0.8000	169	0.4847	180	0.4101
153	1	0.9912	1	1.0000	107	0.4847	2	0.8222
635	58	0.5363	8	0.9000	107	0.4847	97	0.51847
124	5	0.8329	1	1.0000	107	0.4847	115	0.4902
30	80	0.4831	8	0.9000	437	0.1546	174	0.4159
Average	170.7	0.4821	6.1	0.8745	243.9	0.3834	214.5	0.4575
Std	176.1	0.3059	3.9	0.1362	161.7	0.1569	190.2	0.2174

Table 4-1: Prediction score value with its ranking based on different machine learning algorithms

decision tree founds that truly vulnerable files are more similar to each other. In conclusion, all these three aspects indicate that random forest gives the best performance among these four selected classifiers.

The method of analyzing Table 4-2 is different from previous analysis since each ranking value in this table is a relative score comparing to other files in the same fold. We can not compare file ranking or probability estimation between files from different folds directly. When studying this table, I also start with two different angles. First, we inspect vulnerable files in the same fold. As we can see from the table, file 737 and 738 are all in fold 5, and all our selected classifiers assign them similar popularity estimation value. However only Random Forest gives a higher value of estimations to these two files. In most situations, decision trees can find the similarity between vulnerable files in the same fold, but the probability computed is rather lower than other classifiers did. From the perspective of comparing average value of ranking and probability, although random forest did not perform as good as shown in Table 4-1, it still has the lowest average rank among all four classifiers.

To conclude, both these two tables indicate that Random Forest classifier is a good comparative choice for building this prediction model. In the following section, I would describe the testing part of this experiment. For having the conclusion on this research, all four classifiers are used during the testing.

4-4 Test and evaluation

As is shown in the framework description section, all collected data samples are separated into the training set and testing set. While doing the training section, classifiers are designed to learn from the training set. Then we can do testing and evaluation using our trained

Fold	**File**	Naive Bayes		Random Forest		Decision Tree		Logistic Regression	
		Rank	Probability	Rank	Probability	Rank	Probability	Rank	Probability
Fold 1	48	146	3.53e-08	76	0.1100	134	0.5711	79	0.5920
	124	29	0.5529	38	0.1889	134	0.5711	76	0.6000
	30	91	0.0180	133	0.0633	156	0.1153	140	0.2792
Fold 2	153	2	0.9650	13	0.3933	97	0.5146	78	0.4522
	588	109	0.0182	144	0.0033	156	0.0980	140	0.0266
	581	94	0.0259	144	0.0033	156	0.0980	142	0.0227
Fold 3	672	153	2.67e-14	117	0.0067	155	0.1536	155	0.0221
	635	55	0.0844	117	0.0067	66	0.4559	16	0.4272
	710	155	1.21e-52	36	0.0567	66	0.4559	1	0.9866
Fold 4	722	10	0.3951	25	0.1400	154	0.2089	55	0.2734
	734	39	0.3296	125	0.0133	154	0.2089	2	0.4515
Fold 5	737	106	0.2899	5	0.9500	131	0.4962	118	0.5799
	738	105	0.2900	6	0.9467	131	0.4962	119	0.5798
	Average	88.4	0.2	83.7	0.2	125.8	0.37	90.9	0.38
	Std	50.7	0.304	59.4	0.334	32.8	0.189	52.3	0.280

Table 4-2: Probability Estimation Value with its Ranking of Different Machine Learning Algorithms

classifiers. In this section, machine learning algorithms are evaluated by presenting testing results and analyzing the relation between file feature values and classifier estimations.

One purpose of this research is to discover those features which are closely related with the existence of vulnerable files. In answering the question, the study uses two different methods.

4-4-1 Feature Importance Analysis

Random Forest is an averaging algorithm based on several randomized decision trees. Decision tree classifier is trained by learning and setting a series of decision rules inferred from the data features. During the learning procedure, along with the increasing of tree depth, more decision nodes are added to the tree to split smaller size of data samples. So the relative node depth can be used to assess feature importance in a tree. The higher feature importance value is, the more critical feature node is in the tree.

As mentioned in the previous section, I apply principal component analysis (PCA) on original feature space, and the result turns out that machine learning algorithms give better performance on transformed data than on original data. Due to the use of feature extraction method, feature importance values derived from trained Random Forest classifier are corresponding to PCA feature space. PCA components consist of several original features, and If we want to know feature importance for the variables in original feature space, the first step is inverse the importance value from PCA feature space to the original one. Each component in PCA space is generated by a linear combination of several features, also feature importance value is a relative value, linear decomposition process will not destroy the relative relationship between features. Feature importance value ranking is shown in Figure 4-6. To avoid the randomness of Random Forest classifier, I divide the training set into five folds and do training on different folds individually. According to the figure, the most important feature to distinguish between

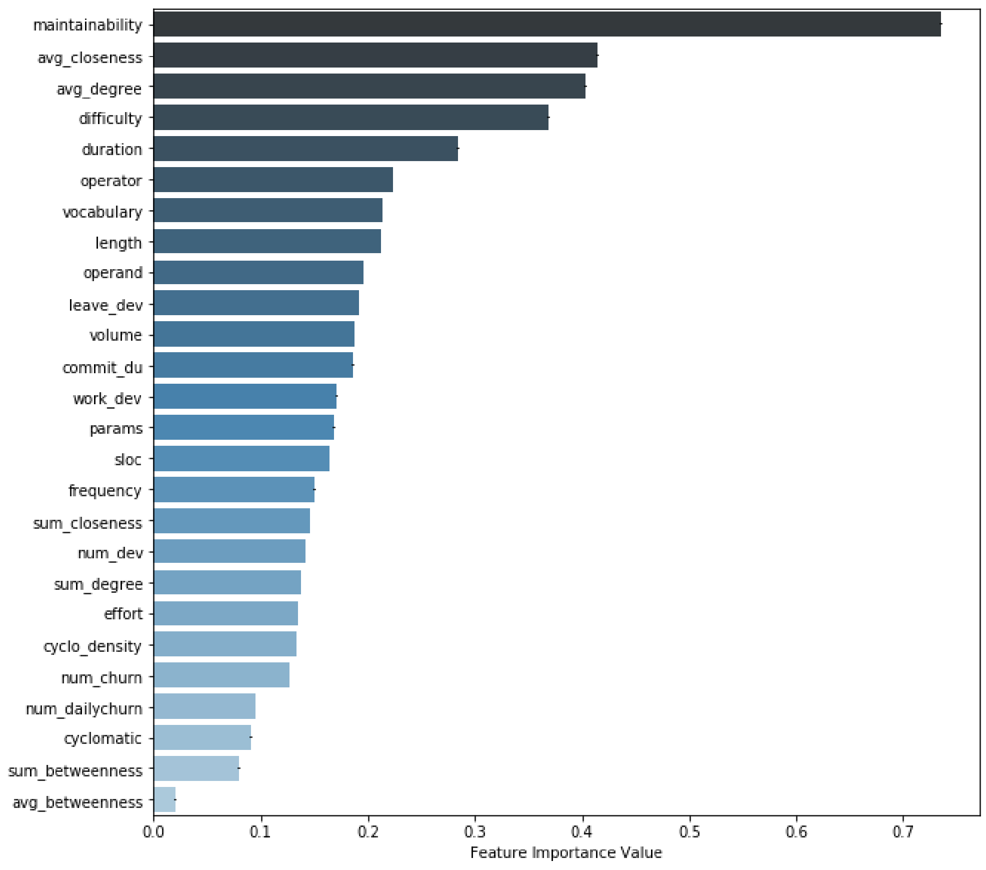


Figure 4-6: Feature importance value retrieved from **Random Forest**

normal and vulnerable files is *maintainability*, which is a derived feature measuring source code complexity, and two features defined to quantify network structure follows. Among top ten features with high importance value, four of them are developer metrics (*avg_closeness, avg_degree, duration and leave_dev*), rest of the feature is all about code complexity metrics (*maintainability, difficulty, operator, vocabulary, length, and operand*).

Feature importance value in Figure 4-6 is the result of learning from the training set. For comparison, I also extract feature importance value from the testing set, and the top 10 important features are displayed in Figure 4-7. As can be seen from the figure, the results learned from the training set and testing set are almost the same. The only difference is that *work_dev* metrics is more important than *leave_dev* for test set.

Through this important features analysis learning from data samples, we can find these causes of web application vulnerabilities distinguished from all the features that I collected. In the next section, I will analyze the classification results from different aspects based on this feature importance analysis.

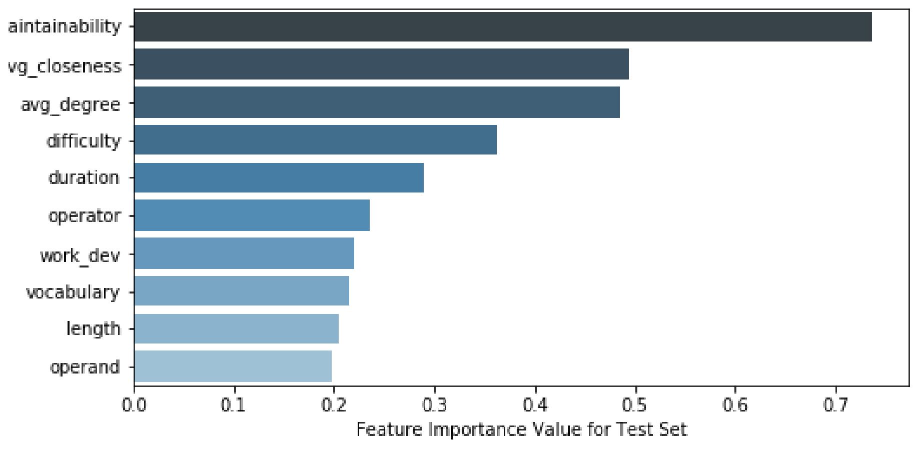


Figure 4-7: Feature Importance Value of Test Set Retrieved from **Random Forest**

4-4-2 Feature Value Analysis

In Figure 4-4, we defined the confusion matrix for this research. True label use 'true' and 'false' to represent 'vulnerable' and 'normal' separately, and for prediction result, 'positive' means the file is labeled to be vulnerable, and 'negative' stands for labeling a file as 'normal' file. To inspect why a file is classified into the normal or vulnerable class based on the feature value analysis, I separate this analysis into several parts.

Vulnerable Files Comparison

In the test set for this research, there are five vulnerable files in total. During the test, Random Forest classifier can find 1 True Vulnerable file while classifying four vulnerable files into the normal class.

Although we made a comparison between classifiers during training part which proves Random Forest gives the best performance, I still use all these four classifiers to test and compare their detected files in order to understand the relation between file features and its label. As we can see from Table 4-3, despite Decision Tree classifier, all other three classifiers provides the same result. It is obvious that file 737 is more likely to be classified into vulnerable class. For knowing the reason, further analysis is made on file 737 by feature importance analysis. From the feature importance analysis, we can find out the top ten relevant features. While introducing the collected dataset in Chapter 3, we have noticed the incoherence that relates to combining data from multiple platforms. The normalization process is done for each platform, feature values of each file show relative degrees comparing to other files in the same platform. Table 4-4 presents the values of these ten features for file 737, it is clear that file 737 has high maintainability value than that of other files in the same platform, and its network metrics values are about the average. Meanwhile, most relevant, important features have relatively low value.

Classifier	TP file index	FN file index	FP file index
Random Forest	737	[588, 30, 581, 48]	[676, 723, 680, 163]
Logistic Discriminant Analysis	30	[588, 737, 581, 48]	[213, 230, 725, 771]
Decision Tree	[30, 48]	[737, 588, 581]	[690, 723, 716, 39]
Naive Bayes	737	[588, 30, 581, 48]	[159, 341]

Table 4-3: TP and FN Files List Generated by Different Classifiers

file index	maintainability	avg_closeness	avg_degree	difficulty	duration	operator	vocabulary	length	work_dev	operand
737	0.8353	0.5000	0.6703	0.0309	0.0624	0.0046	0.0236	0.0047	0.01	0.0205

Table 4-4: Vulnerable file feature value list

From this individual file analysis, we can only know the characteristics of it compared to other files in the same platform. The similarity shared by all vulnerable files is something we also need to consider about. In Table 4-5, presenting all False Negative results with their feature values. True positive results and false negative results are all labeled vulnerable files, based on this; first analysis perspective is to find a similarity between them. Directly shown in the table, all vulnerable files have a small size of active developers. Code complexity metric value varies from file to file, from the table we can not tell whether a more complex file would have a higher probability of being vulnerable. After finding the similarity, we also need to know the difference between them considering not all vulnerable files are detected by our prediction model. False positive files preserve higher value on most code complexity metrics, and if *avg_closeness* and *avg_degree* of a file are both larger than that of the true positive file, this file is more likely to be normal.

Comparison between Normal Files and Vulnerable Files

In this section, Normal file means the file is actually without any vulnerabilities, and Vulnerable file represents a file with vulnerabilities. Moreover, a positive file stands for a file which is labeled as vulnerable by our designed prediction model and negative file means it is labeled as normal.

For showing the distribution of feature value, we use the boxplot method to visualize the result. In the figure, all positive files share similar code complexity feature values, while these features can distinguish positive and negative files. So derived from these figures by comparing all positive files with negative files, code complexity metrics play an important role in helping classifier to make predictions, although the result is not that good enough. While through comparing true files with false files, the figure also shows only developer metrics, including

file index	maintainability	avg_closeness	avg_degree	difficulty	duration	operator	vocabulary	length	work_dev	operand
588	0.6929	0.1875	0.1429	0.3206	0.0607	0.0665	0.1019	0.0609	0.25	0.0723
30	0.3172	0.5789	0.6117	0.7309	0.8476	0.7333	0.4038	0.3783	0.01	0.3867
581	0.7191	0.1875	0.1428	0.0332	0.00	0.0051	0.0204	0.0050	0.01	0.3867
48	0.3894	0.6842	0.6808	0.3933	0.7363	0.5000	0.0938	0.0586	0.01	0.0138

Table 4-5: Vulnerable File Feature Value List

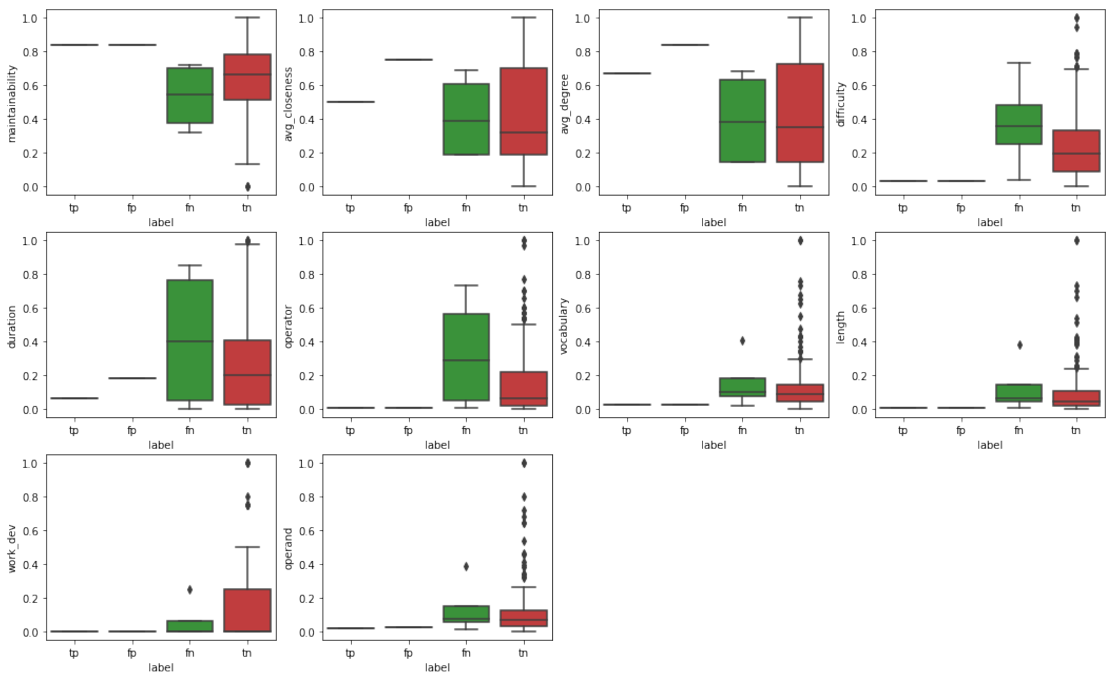
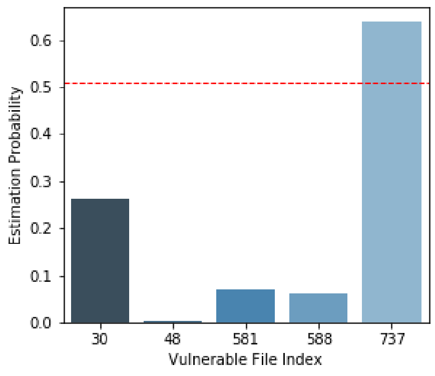


Figure 4-8: Value Distribution of Top 10 Important Features

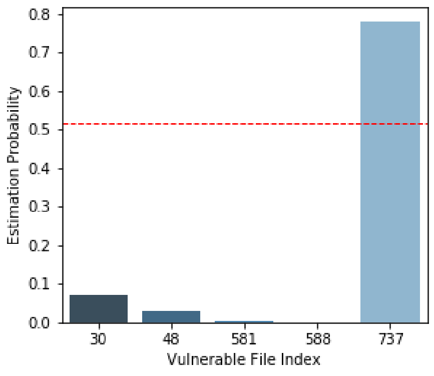
avg_closeness, *duration* and *avg_degree*, are able to distinguish false positive file and true positive file. An initial conclusion can be drawn from the figure is that developer metrics including developer behavior metrics, and developer network metrics are more important for distinguishing normal and vulnerable files. Based on this initial conclusion, further researches are done in section 4-4-4 to prove the conclusion by using another method.

4-4-3 Probability Estimation Analysis

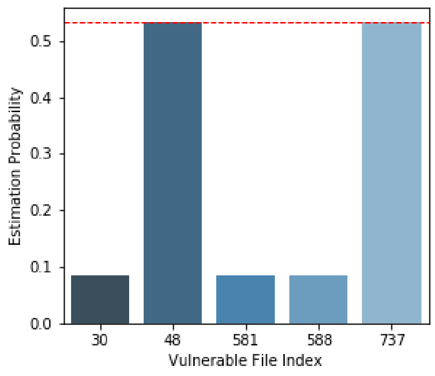
We focus on analyzing feature values to discover the cause of having web application vulnerabilities in previous sections. In this section, I focus on studying the estimation probability computed by different classifiers. In Figure 4-9, the x-axis shows the index of all vulnerable files in the test set, and the y-axis displays the estimation probability. Also, for understanding how a classifier makes decisions on labeling, the figure also shows the decision boundary determined by these classifiers. Random Forest assigns high probability estimation on file 737, but fail to distinguish other four vulnerable files from normal files. The situation faced by Naive Bayes is similar to Random Forest, but Naive Bayes is more likely to find something abnormal on file 30. This plotting also has some disadvantages, since the figure cannot show the probability distribution for normal files. We cannot determine the expense of lowering the decision boundary from this figure, so we are not able to conclude that Logistic Discriminant Analysis is better than other classifiers. However, this figure can instruct us which file might be valuable look into further. Like file 581 and 588, all classifiers assign both of them with low probability estimation. According to my investigation, both of them are from a new platform that has been recently updated, and only this platform is developed based on **Node.js** framework. This result inspires a new direction for the future work on studying the relationship between



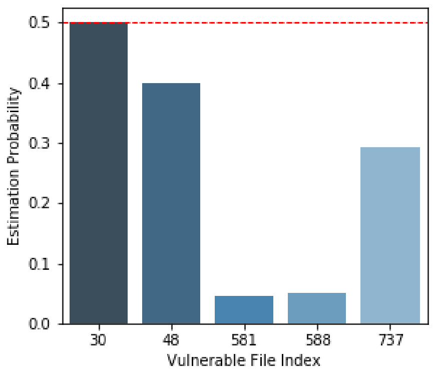
(a) **Naive Bayes**



(b) **Random Forest**



(c) **Decision Tree**



(d) **Logistic Discriminant Analysis**

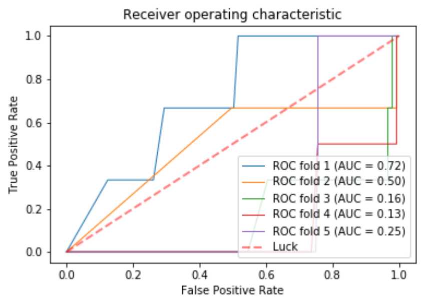
Figure 4-9: Probability Estimation Result of all Vulnerable Files by Using Different Classifiers

developer metrics with web application vulnerabilities.

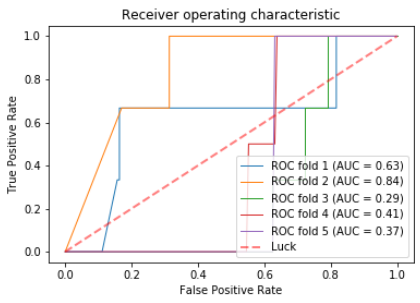
4-4-4 Comparison between feature set

At the beginning of this research, I raise a research question about studying whether code complexity, developer behavior and developer network features are important for predicting web application vulnerabilities or not. For answering this research question, I divided the collected data into three individual datasets, and they include features about developer behavior, developer network and code complexity separately. I use three datasets as the input of designed prediction model in turns, and for evaluation, I choose to use ROC curves to show the result.

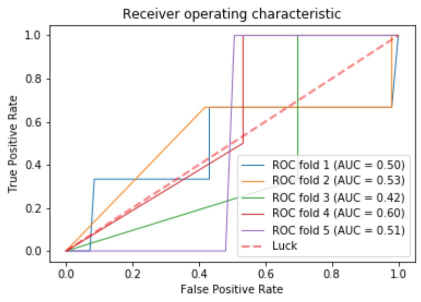
All results are shown in Figure 4-11, to make the comparison, I set 0.7 as the ideal true positive rate. For Code Complexity dataset, the false positive rate would be greater than 0.6 in most folds. However, developer behavior dataset results in lower false positive rates than code complexity. Moreover, if we set 0.2 as the ideal number of false positive rate, developer-related



(a) **Code Complexity**



(b) **Developer Behavior**



(c) **Developer Network**

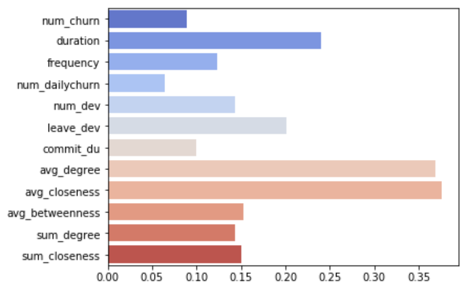
Figure 4-10: ROC curve and AUC value by using different feature sets

metrics can result in higher true positive rate than dataset with code related metrics. Also, the average AUC value also indicates the better performance brought by using developer related metrics. From this part of the research, we can conclude that developer related metrics are more important than code complexity metrics when building a web vulnerability prediction model.

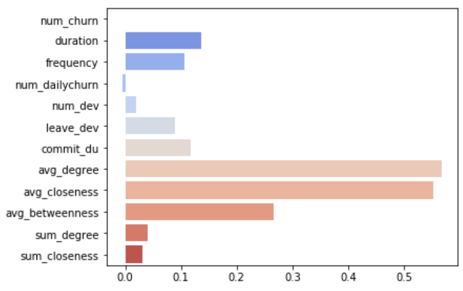
4-4-5 Investigation on False Positive files

In this research, Checkmarx's detection result is settled to be the ground truth. It is possible that this designed prediction model is able to discover some vulnerabilities existed in the files that have not been detected by Checkmarx. To investigate whether the research can discover vulnerabilities that Checkmarx cannot detect, with the support from developer team, I select several false positive files from the prediction result, including file 39, 163 and 230.

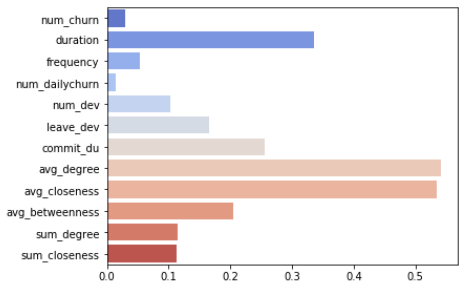
Before discussing with developers about the false positive results made by the prediction model, I did an analysis based on collecting feature contributions extracted from the Random Forest classifier. It is common known that the Random Forest classifier is a black-box method and it is impossible to interpret the whole decision process. As a tree-based classifier, prediction value calculation is defined in Equation 4-2, where each k represents a feature. The larger the *contribution* value is, the feature would have more influence on *prediction*.



(a) **File 39**



(b) **File 163**



(c) **File 230**

Figure 4-11: Feature Contribution of labelling files

$$prediction = c_{initial} + \sum_{k=1}^{K} contribution(x, k) \tag{4-2}$$

When comparing *contribution* value for predicting each source code file, the result is similar to the result shown in feature Importance Analysis in section 4-4-1, developer network metrics, like *avg_degree* and *avg_closeness*, also have high contribution value According to the information provided by developer team, file 163 is a source code file created by the third-party team. A third-party file can contain more potential factors other than the aspects we considered that can result in producing vulnerabilities. This information gives inspiration to the future work that considering the source of files is also useful for this area of research.

The other two files are all related with providing information to page viewers. The similarity shared by these files also highlight a new direction that the functions realized in the file might also related with the probability of having vulnerabilities. For example, a function to build a message box is more likely to have vulnerabilities than a static web page without any input form.

Figure 4-11 shows the feature contributions for labeling different files, which is clear that file-based network metrics like $avg_{d}egree$ and $avg_{c}loseness$ have made large contributions for labeling a file. For further study, we can collect developer network metrics from other resources instead of introducing network theory, in order to know whether the actual developer team structure and collaboration is correspond to metric values.

4-5 Discussion

Previous analysis on the prediction results can be divided into two main parts. During training phases, the analysis helps the research to build a prediction model with good performance. While in testing part, I analyze the prediction results and point out how a trained classifier makes decisions on classifying each file. These analysis proves the importance of considering developer related metrics when building a prediction model, and these developer related metrics are more useful than code complexity metrics.

However, this research also has several limitations. Firstly, the number of vulnerable files is limited, because we only take JavaScript files into account. With only 13 vulnerable files in nearly 1000 normal files, it is vital to decide how to divide training and testing set properly and introduce imbalanced learning method into this research. Secondly, file ground truth labels are not reliable enough due to the existence of false positive labels. For improving the performance of this designed prediction model, it might be useful to label each file by using reliable vulnerability check result. Thirdly, the performance of this prediction model is not good enough, although we have done data processing and parameter tuning in this research.

Chapter 5

DISCUSSION

This Chapter concludes this research on building a prediction model for discovering web application vulnerable files. In this Chapter, we do recall on the whole experiment process by pointing out how to execute the experiments and what we have found during the research in the beginning, then discussing the experimental methods, results and limitations of this research. At last, answering three research questions raised in Chapter 1. Also, on the basis of research limitation, promoting new direction and suggestions to the future works.

5-1 Research Work Recap

This research is focusing on building a prediction model for detecting vulnerable files. All collected data are originally acquired from GitHub repository. GitHub repository consists two main kind of resource, including source code files and commitment history log files. Previous researches on building prediction models for detecting vulnerable files considers only the quality of the code, without taking human factors into account. This research starts from a new perspective of introducing developer related metrics into input feature space, and using traditional machine learning algorithms to learn from input data to predict sample data. The whole experiment procedure includes three parts.

5-1-1 Data Collection

Data collection process not only involves determining the features that is useful for the prediction model, but also includes where to collect designed features. Both of them could influence the performance of the prediction model. Due to this reason, we put efforts on studying what resources are useful for this project and how to use them. By using *npm* project "PLATO", I am able to extract all code complexity metrics from each development package. For developer related metrics, I can only use GitHub commitment record to calculate some static metrics, for measuring developer team network structure, I introduce network theory into this research. Moreover, there is no guarantee that the vulnerability check results could provide the ground-truth result to label our files, which have false vulnerable files records, and it might influence the performance. We did an initial data analysis on the raw data by visualizing the feature value distributions of normal files and vulnerable files separately. The figures present that some features show the distinctions between these two kinds of files. Based on the visualization, I raised several suggestions to developers and managers that can help to reduce the risk of having vulnerabilities by adjusting the settlement of their development teams.

5-1-2 Classifier Training

Building a machine learning system for this prediction model is also count for a large part my research. In the beginning, I notice that this research should pay attention to the imbalance of the collected dataset which was inspired by several papers about software defects detection. Considering that I have all labeled data for training, so only supervised learning algorithms are considered in this research. For imbalanced data learning, resampling method is used to improve the performance of prediction model, like many other research on imbalanced data learning, I used SMOTE method to do oversampling on training set. When selecting machine learning algorithms to build the prediction model, I take both black-box and white-box methods into account, and decided to use Naive bayes, Random Forest, Decision Tree and LDA classifiers. Methods for imbalanced learning evaluation are different from other balanced learning evaluation, and most researches used ROC curve and AUC value to evaluate and compare the performance. To show the details about prediction probabilities of each file, I listed prediction probabilities and rankings for vulnerable files. By comparing the rankings generated by different classifiers, it turns out that Random Forest gives the best performance.

5-1-3 Testing and Evaluation

For testing and evaluation on this prediction model, I select several different aspects to analyze the prediction results. Random Forest is a tree-based classifier, so it can compute the feature importance value according to the decisions made by each tree branch. The ranking of feature importance value presents that developer network metrics is important for making decisions, but half of the metrics are related with code complexity. Since there are limited number of vulnerable files in test set, I used a table to show their predicted label. Noticing that file 737 is more likely to be classified into vulnerable class, I display the feature values of this file and try to find patterns of a vulnerable file. Despite before, I also do the comparison between some normal files and vulnerable files and the result turns out that developer related metrics are more useful for distinguishing true positive and false positive results. Then, to answer the research question raised in Chapter 1, I do some experiments by using different feature sets and compare the performance. The result clearly indicate that using developer related metrics are performing better than code related metrics on predicting vulnerable files. At last, in order to find some new directions for future work, I spend some time to study the false positive files. According to my discovery, a third-party file is always misclassified to vulnerable file, which might be useful to add a metric which can present the source of a code file.

5-2 Conclusion

To draw conlcusions, we first recall our research question.

- How to quantify developer works on each source code file from developer working log files?
- What suggestions can be made to adjust developer behaviors based on collected metrics by using data analysis method?
- Is code complexity, developer behavior and developer network metrics important for detecting web application vulnerabilities? Which one is most important to distinguish vulnerable files from normal files?

5-2-1 Research Question 1

- How to quantify developer works on each source code file from developer working log files?

In this research, several static metrics are defined to measure developer's working behaviour. I used the information included in GitHub repository. Details for calculation are shown in Chapter 3.

5-2-2 Research Question 2

- What suggestions can be made to adjust developer behaviors based on collected metrics by using data analysis method?

In Chapter 3, I visualize the density distribution of some features and suggestions are made in the conclusion part. Based on the information provided in the figure, I am able to give some advice to platform developers. Also, the feature importance values can also give indication to the way of adjusting development work.

5-2-3 Research Question 3

- Is code complexity, developer behavior and developer network metrics important for detecting web application vulnerabilities? Which one is most important to distinguish vulnerable files from normal files?

According to the results I found during the experiment, it shows that developer related metrics are more important than code related metrics. But code complexity metrics are also important for predicting vulnerable files, as is shown in the feature importance value results.

5-3 Future Work

5-3-1 Exploring features

The features we collected in this research are all static features. Adding version label to each file. Inspired from predicting third-party code files to be vulnerable, it is also useful to add another feature to distinguish whether the file is developed by internal groups or not. Also, classifying files according to their usages might also help. Moreover, all the metrics that I collected in this research are static. So further work could be done to collect some dynamic metrics to measure developer works. For example, we can collect metrics every once in a while and add time stamp to these metrics.

5-3-2 Labelling source code files

In this research, the ground truth label is determined by Checkmarx (a vulnerability detection tool) result. As an automatic detection tool, it is unavoidable that Checkmarx's vulnerability report would have false positive, in other words, detected vulnerability actually do not exist. For improving the performance of this prediction model, learning procedure is deterministic. If we can ensure true labels do not contain false positive result, the prediction model can learn things more precisely.

5-3-3 Unsupervised learning

This research only choose supervised learning to build the prediction model, because of considering that the labeled data is accessible. If it was difficult to access the ground-truth label of training data samples, it would be a wise choice to use unsupervised learning algorithms. Some papers about software defects also mentioned about using semi-supervised learning method to do prediction, which also worth to try.

Bibliography

[1] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, “Supervised machine learning: A review of classification techniques,” *Emerging artificial intelligence applications in computer engineering*, vol. 160, pp. 3–24, 2007.

[2] H. He and E. A. Garcia, “Learning from imbalanced data,” *IEEE Transactions on Knowledge & Data Engineering*, no. 9, pp. 1263–1284, 2008.

[3] S. Wang and X. Yao, “Using class imbalance learning for software defect prediction,” *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[4] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[5] H. He, Y. Bai, E. A. Garcia, and S. Li, “Adasyn: Adaptive synthetic sampling approach for imbalanced learning,” in *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pp. 1322–1328, IEEE, 2008.

[6] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer, “Smoteboost: Improving prediction of the minority class in boosting,” in *European conference on principles of data mining and knowledge discovery*, pp. 107–119, Springer, 2003.

[7] T. Menzies, J. Greenwald, and A. Frank, “Data mining static code attributes to learn defect predictors,” *IEEE transactions on software engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[8] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, “Identifying the characteristics of vulnerable code changes: An empirical study,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 257–268, ACM, 2014.

[9] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[10] Y. Shin and L. Williams, "Is complexity really the enemy of software security?," in *Proceedings of the 4th ACM workshop on Quality of protection*, pp. 47–50, ACM, 2008.

[11] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[12] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, pp. 284–292, ACM, 2005.

[13] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 13–23, ACM, 2008.

[14] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[15] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Information and Software Technology*, vol. 54, no. 5, pp. 467–478, 2012.

[16] O. W. A. S. Project, "Xss (cross site scripting) prevention cheat sheet," 2018.

[17] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis.," in *NDSS*, vol. 2007, p. 12, 2007.

[18] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Computer Security Applications Conference, 21st Annual*, pp. 9–pp, IEEE, 2005.

[19] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *IFIP International Information Security Conference*, pp. 295–307, Springer, 2005.

[20] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 387–401, IEEE, 2008.

[21] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, pp. 6–pp, IEEE, 2006.

[22] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 6, pp. 688–707, 2015.

[23] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 642–651, IEEE Press, 2013.

[24] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, pp. 92–101, ACM, 2014.

[25] M. H. Halstead *et al.*, *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., New York, NY, 1977.

[26] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[27] C. D. D. L. Jesse Harlin, Jarrod Overson, "es6-plato." `https://github.com/deedubs/es6-plato`, 2012.

[28] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories.* 2018.